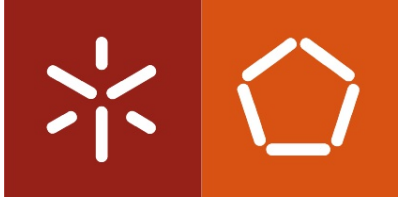
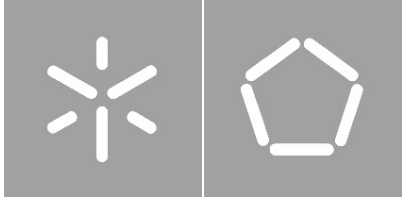


Universidade do Minho
Escola de Engenharia

Leonel João Fernandes Braga

Web Browser Access to Cryptographic Hardware

Outubro de 2012



Universidade do Minho
Escola de Engenharia

Leonel João Fernandes Braga

Web Browser Access to Cryptographic Harware

Tese de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
Doutor Vítor Francisco Fonte

Supervisão na empresa de
Engenheiro Renato Portela

Outubro de 2012

DECLARAÇÃO

Nome: Leonel João Fernandes Braga

Endereço electrónico: pg17311@alunos.uminho.pt

Telemóvel: +351 932 852 847

Número do Cartão do Cidadão: 13321692 - ZZ9

Título dissertação □/tese □

PT: Acesso a Hardware Criptográfico via Web Browser

EN: Web Browser Access to Cryptographic Hardware

Orientador: Vitor Francisco Fonte

Supervisor na empresa: Renato Portela

Ano de conclusão: 2012

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento: Mestrado em Engenharia Informática

Universidade do Minho, 31/10/2012

Assinatura: Leonel Braga

Acknowledgments

I could not conclude this work without acknowledge all the support, time, and understanding of all the people who have been around me during this phase and during my journey of life. I am sure that without them everything would be much more difficult, and the success would be harder to achieve.

First of all, I want to thank to my supervisor Professor Victor Fonte for being so helpful and supportive. His guidance certainly improved my work and my knowledge as well. I want also to thank to Engenheiro Renato Portela from MULTICERT for enlightening me when I was more doubtful.

A special thanks to MULTICERT for letting me enrol in this project: it made me grow professionally and enhanced my knowledge.

I want also to thank the *Firebreath* community for clarifying all the doubts I had. Congratulations for your great work as well.

In this context, there is one person to whom I could not be more grateful: Pedro, thank you for your help and patience, even when I had lots of questions. I am also grateful for the discussions I had with Pedro and Ulisses: they gave me lots of ideas of how I could improve my work. I want to thank Vasco for introducing me to *jQuery* and for providing his experience building beautiful websites. Thanks for your friendship as well.

A humble and sincere thanks to Joana's Family for giving me a second home.

A special and kindly thanks to my family, to whom I will always be in debt and grateful for all the opportunities and support they gave me in each step of my life. Thank you, my little sister Joana for your help.

At last, but not least, I wish to express my gratitude to Joana for kindly supporting me with her sympathy in every moment.

Abstract

Web Browser Access to Cryptographic Hardware

Cryptographic hardware such as Smart Cards (SCs) is being deployed globally in an increasingly broader spectrum of information services, credit and debit banking cards being a pervasive example of this trend. At the national level, the Portuguese Citizenship Card (PCC) is a high profile example of this technology, allowing users to do online authentication at the government Internet-based services. Despite this increasingly common scenario, web browsers — expect those from the Mozilla Foundation — still have limitations when accessing cryptographic hardware due to the absence of a standard — or at least uniform — mechanism accessible to the programming logic embeddable in web pages.

In this project we propose a new mechanism to address such limitations, which will expose SCs to web applications in a clean and uniform way among web browsers. This mechanism is formed by two main elements: a web browser plugin, and a JavaScript (JS) Application Programming Interface (API). The plugin will be in charge of connecting the web browser to the SC. The JS API, accessible through the web browser plugin, will expose the SC features to web applications.

With the conclusion of this project we managed to successfully create a web browser plugin which allows web applications to access SC related features, such as the creation of Digital Signature (DS). In our tests we were able to use and check all the features of the plugin across several web browsers (Google Chrome, Internet Explorer, and Firefox) and operating systems (OSs) (Ubuntu, OS X, Windows). The security analysis that we performed helped us identify the likelihood of possible attacks which could led malicious agents to gain access to the users' computers, or get their personal and sensitive data.

Keywords: Web Browser Plugin, Cryptography, Smart Card, Public-Key Cryptography Standards, PKCS#11, Web Applications

Resumo

Acesso a Hardware Criptográfico via Web Browser

O hardware criptográfico, como é o caso dos Smart Cards (SCs), tem vindo a ser utilizado num espectro cada vez mais amplo de serviços de informação, sendo os cartões de crédito e de débito um exemplo desta tendência. A nível nacional, o Cartão de Cidadão constitui um exemplo notável de aplicação desta tecnologia, permitindo aos utilizadores efetuar a sua autenticação online em serviços do governo presentes na Internet. Apesar destes cenários serem cada vez mais comuns, os browsers web — à excepção daqueles provenientes da Fundação Mozilla — possuem limitações no acesso ao hardware criptográfico, devido à inexistência de um mecanismo padrão — ou pelo menos uniforme — disponível para a programação de aplicação web.

Neste projecto, propõe-se um novo mecanismo para resolver as limitações citadas, através de uma exposição dos SCs a aplicações web de uma forma clara e uniforme entre os browsers web . Este mecanismo é composto por dois elementos principais: um plugin para o browser web e uma Application Programming Interface (API) em JavaScript (JS). A ligação entre o browser web e o SC é estabelecida pelo plugin mencionado. A interface em JS, acessível através do plugin do browser web , expõe as características do SC às aplicações web.

Neste projecto desenvolveu-se com sucesso um plugin para browsers web que permite o acesso das aplicações web às funcionalidades do SC, como a criação de uma Assinatura Digital. Nos testes desenvolvidos, foi possível utilizar e verificar todas as funcionalidades do plugin em vários browsers web (Google Chrome, Internet Explorer, and Firefox) e sistemas operativos (Ubuntu, OS X, Windows). A análise de segurança realizada permitiu identificar a possibilidade de existência de locais de ataque que agentes maliciosos podem potencialmente utilizar para aceder aos computadores dos utilizadores, ou obter os seus dados pessoais.

Keywords: Web Browser Plugin, Criptofrafia, Smart Card, Public-Key Cryptography Standards, PKCS#11, Aplicações Web

Contents

List of Figures

List of Tables

List of Examples

Acronyms

APDU	Application Protocol Data Unit
API	Application Programming Interface
CA	Certificate Authority
CDSA	Common Data Security Architecture
CSSM	Common Security Services Manager
DS	Digital Signature
DOM	Document Object Model
DER	Distinguished Encoding Rules
GPG	GNU Privacy Guard
IDE	Integrated Development Environment
JS	JavaScript
JVM	Java Virtual Machine
JCWS	Java Card Web Servlet
LOC	Lines of Code
MAC	Message Authentication Code
MY	Man Years
NPAPI	Netscape Plugin Application Programming Interface
NaCl	Native Client

OS operating system

PPAPI Pepper Plugin API

PIN Personal Identification Number

PCC Portuguese Citizenship Card

PKC Public-key Cryptography

PKI Public-key Infrastructure

PKCS #11 Public-Key Cryptography Standards #11

SC Smart Card

SCR Smart Card Reader

STL Standard Template Library

SSL Secure Sockets Layer

TLS Transport Layer Security

W3C World Wide Web Consortium

Chapter 1

Introduction

Cryptographic hardware such as Smart Cards (SCs) are present in several services of our everyday life, such as public transportation and telecommunications. These small devices allow us to carry our personal information in a portable and secure way.

A typical SC has an integrated circuit, embedded in the card body, capable of transmit, store and process data [Rankl and Effing, 2004]. In its design we can identity three layers, from hardware to software, which can control the access to the data, protecting it against manipulation and unauthorized access [Selimis et al., 2009, Rankl and Effing, 2004]. At the bottommost layer is the hardware, transparent to users, with the following parts: a *microcontroller*, *RAM*, *ROM*, *EEPROM*, a *Coprocessor*, and input/output interfaces. The operating system (OS) is in the middle and manages the resources. The topmost layer corresponds to the Smart Card applications [Selimis et al., 2009].

The tamper-resistant properties of SCs make them an ideal device to use in Public-key Cryptography (PKC). SCs can safely store either public key certificates and private keys, and they have built-in cryptographic algorithms to encrypt and decrypt data, and even to create Digital Signatures (DSs) [Adams and Lloyd, 2003]. The multifactor entity identification is also one of the major advantages of using SCs in PKC. A multifactor identification scheme requires users to have a valid public key certificate stored inside a SC — the what-you-have factor — and this user must know the Personal Identification Number (PIN) needed to unlock the SC — the what-you-know-factor. This scheme can identify — with a high level of assurance — users of a given system, addressing some identity theft attacks related to password-based authentication [Aussel, 2007, Lu and Ali, 2010].

Thus, the adoption of SCs is a natural step in many services to enhance the overall system's security. In Health Care a SC can be used to safely store patients' records. Mobile communications corporations use SCs to identify its clients, every time they use their cell phone or

their *USB* modem for wireless Internet. Some online services, such as e-Government or home banking, are also using SCs to protect the end-user identity credentials [Sauveron, 2009].

At the national level, the Portuguese Citizenship Card (PCC) is a high profile example of this technology. Currently, each Portuguese citizen owning one of these SCs can authenticate at online e-Government services digitally signing files, using both the authentication and signing certificates present in this device [Agência para a Modernização Administrativa, 2008]. This operation has the same legal value as a hand-written signature due to the cryptographic properties of a DS [Adams and Lloyd, 2003]:

- authentication, it is possible to identify precisely who created the signature;
- integrity, any alteration to the document violates the signature;
- non-repudiation, the signer cannot deny the signature.

1.1 Motivation

Smart cards are being globally deployed in an increasingly broader spectrum of information services. However, web browsers still have limitations when accessing SCs due to the absence of a common standard — or at least an uniform — mechanism accessible to the programming logic embeddable in web pages.

The Public-Key Cryptography Standards #11 (PKCS #11)[1], also known as *Cryptoki*, is a de facto standard created by the *RSA Laboratories*[2] which defines a uniform and cross-platform Application Programming Interface (API) to SC. This standard specifies in an object-based approach a high level mechanism to either inspect the cryptographic contents, or to perform cryptographic operations on SCs, among many other functions.

Usually the vendors of SCs compliant with this standard, supply a module that can be used by software developers to connect applications with the cryptographic device. Besides the modules deployed by software vendors, there is a community-driven open-source project named *OpenSC*[3] that supplies libraries and utilities to work with SCs. This software is available for all platforms and has support for many SCs, such as the Portuguese and Estonian Citizenship Cards[4]. Among all the utilities and libraries there is also a PKCS #11 module that can be used in all major OSs to access SC.

[1] http://www.rsa.com/rsalabs/node.asp?id=2133
[2] http://www.rsa.com/
[3] http://www.opensc-project.org/opensc
[4] http://www.opensc-project.org/opensc/wiki/SupportedHardware

In order to access SCs, web application developers have been deploying custom software — like *Java Applets* and *ActiveX Controls* — that take advantage of the SC. Usually these solutions lack portability, forcing users to work with a particular — often unfamiliar — web browser and OS in order to successfully access each particular service, and thus compromising the cross-platform compatibility of a web application.

The absence of a standard mechanism to access SC forces web application users to resort to distinct software packages, according to which specific SC and web application they use. This situation can increase the probability of a security breach, because there are several software packages being use. An attacker can take advantage of a vulnerability in a plugin, either by exploiting a flaw in its construction or either by using a known flaw in the technology it was built — like *ActiveX* controls. The opportunity for phishing schemes may also increase. Attackers can lure users to install ill-intentioned plugins similar to the ones they use on a trustworthy web application, in order to get local, private, important and personal data from users.

1.2 Goals

The main goal of this project is to create a mechanism which connects web applications and Smart Cards. The focus of our work is to expose SCs to web applications in a clean and uniform way, among OSs and web browsers.

The mechanism that we propose introduces two new connectivity layers between web applications and SCs in web browsers. The first layer of this mechanism is a plugin which enables a web browser with SC connectivity capabilities. These Smart Card capabilities will then be exposed by the plugin to web applications through a JavaScript (JS) API — the second layer.

Figure 1.1 describes at a high level view how this mechanism must work. In order to a web application access a SC, its clients must have installed on the web browser this new plugin. The client side of the web application will perform JS requests to the plugin in order to communicate with the SC. The plugin will interact with the SC using a Smart Card access library.

As mentioned before, the major goal for this project is to develop a clean and uniform mechanism that exposes SCs to web applications. To accomplish this goal we will develop a web browser plugin that acts as frontend to the available features in SCs. We intent that this mechanism can be easily ported to several web browsers, and that its functionalities be browser-independent. In particular, due to MULTICERT requirements — a stakeholder

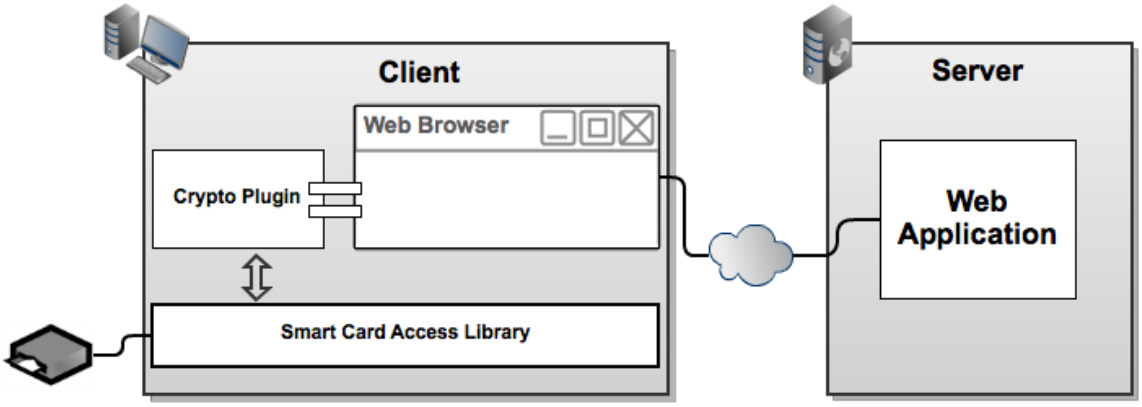


Figure 1.1: Mechanism Overall Structure

of this project — the plugin must be compatible with the following web browsers: Google Chrome[5], Microsoft Internet Explorer[6].

The mechanism should enable a web application to:

- inspect if there is a SC present in the computer;
- get notifications anytime a SC is inserted or removed from the computer;
- get information about SCs, public-key certificates and available mechanisms on the SC;
- create DSs either of data or files.

We will use the PKCS #11 standard as the SC access library, because it can be found in all major OSs — which can be very usefull to build a cross-platform plugin — and MULTICERT[7] requires it.

The PCC will be the proof of concept of this work due to the following reasons:

- any Portuguese citizen has easy access to one of this devices, which makes it easier to test the plugin with a SC that is being broadly used in Portugal;
- the Portuguese Government supplies a PKCS #11 module and this SC is supported by *OpenSC*;
- among its features, the capability of digitally sign documents, or any other kind of data is something that will be interesting to provide.

[5] https://www.google.com/intl/en/chrome/browser
[6] http://windows.microsoft.com/en-us/internet-explorer/products/ie/home
[7] https://www.multicert.com/home

The final task of this project is to perform a exploratory security analysis of the plugin. With this analysis we intent to:

- find weaken spots that can be exploited by attackers in order to access or manipulate the user data present in his SC or even in his computer;
- argue which measures can be put in place in order to address the problems we may found;

1.3 Contribution

In this project we successfully developed a web browser plugin that exposes to web applications SCs through a JS API. At this point the plugin is able to:

- list the available devices, as well as theirs details;
- get a list of available cryptographic mechanisms;
- get available private and public keys, as well as their details;
- get available public key certificates, as well as their details;
- fire an event to the web application whenever a device is either inserted or removed;
- create DSs;
- create digests.

In our tests we used the PCC and the PKCS #11 module supplied by the Portuguese Government, but we expect that any Smart Card providing a *p11* module will be compatible with the plugin, as well as any other SC supported by *OpenSC*.

The plugin was successfully tested under the following platforms and web browsers:

Google Chrome Microsoft Windows XP Professional SP3, Mac OS X Snow Leopard, LUbuntu 12.04

Mozilla Firefox Microsoft Windows XP Professional SP3, Mac OS X Snow Leopard, LUbuntu 12.04

Microsoft Internet Explorer Microsoft Windows XP Professional SP3

We expect that any other browser which supports the Netscape Plugin Application Programming Interface (NPAPI) architecture will also support our plugin, because we did not implement web browser specific features.

In this project we also briefly analysed the plugin security. In this analysis we used static tools to inspect the source code in order to find problems like *buffer overflows*. We used Attack Trees [Schneier, 1999] to create a model of what kind of goals an attacker may want to achieve attacking the plugin. These models helped us discuss the attacks that can be performed against each goal, and this devising/propose strategies intended to mitigate the related risks. For each attack we identified which counter-measures to address such attack, or, at least, reduce the vulnerability to an admissible level. Finally, we also measure the code maintainability using the model by SIG [Heitlager et al., 2007], because it is easier to analyse a source code for security vulnerabilities when it is easy to maintain it [Seacord, 2008].

1.4 Dissertation Outline

In the next lines we a describe the document's structure, where the reader can find the major topics of each chapter:

Chapter 2 covers the current mechanisms to access SCs and develop web browser plugins, and to protect both users and applications. We also review in this chapter the related work.

Chapter 3 describes the development of the plugin, from its design to its implementation and usage. There is also a *Firebreath* glimpse of how one can use this framework to create a web browser plugin.

Chapter 4 presents a security analysis of the plugin. In this chapter we show what tools we used to check the plugin security, and we discuss their results.

Chapter 5 covers the project final remarks, where we discuss the fulfillment of our original goals. We finalize this chapter and conclude this document with what we think must be the guidelines for future releases of the mechanism we proposed.

Appendix A presents the documentation of the JS interface of the plugin, and a set of tables containing the available data for each one of the PKCS #11 objects.

Chapter 2

Related Work

Reviewing the current available solutions for a problem is a first step towards a successful work: it allow us to learn with what others did (with their mistakes and achievements) and to identify which improvements can be made in order to build a better and distinct solution.

In this chapter we present some relevant projects and technologies, from industry to academics and online communities, that tried to enhance the Smart Card (SC) capabilities in web applications, and operating systems (OSs). Then, we discuss the current techniques to access SCs through applications, and present the available methods to develop web brower plugins. The last section of this chapter introduces some concepts about the current techniques to protect users and applications from malicious agents, such as *Code Signing* and *Application Sandboxing*.

2.1 Web Browser Access to Smart Cards

Smart Cards are being applied to many services due to their portability and security. Nevertheless, web browsers still have some limitations in working with them. They do not provide to web applications any mechanism that exposes SCs functionalities. This has caused many web applications to develop their own software, in order to communicate with SCs. All these non-standard solutions follow the exact same pattern, where developers publish either Java Applets[1], or web browser plugins using the *NPAPI*[2] Application Programming Interface (API) for *Netscape*-based browsers, or *ActiveX* for Microsoft Internet Explorer [Sachdeva et al., 2009].

[1] http://www.developer.com/java/other/article.php/3587361/Java-Applet-for-Signing-with-a-Smart-Card.htm

[2] https://wiki.mozilla.org/NPAPI

The only web browser which currently allows web applications to make use of cryptographic hardware is the Mozilla Firefox. The Mozilla *JavaScript Crypto Library*[3] library provides methods that allow web pages to access cryptographic related services, such as: handling SC events — smart card insertion and smart card removal; authenticate users and sign text with the certificates stored in a SC. This cryptographic library is a specific Mozilla extension, and it cannot be adopted in other web browsers, because it is not a web browser plugin but a specific feature of *Mozilla* web browsers.

In the official development website of the Google Chrome web browser we can find in the project's roadmap the ticket *"Investigate the possibility of supporting digital signing with PKI"*[4], where users and developers discuss the introduction of mechanisms to support the creation of Digital Signature (DS) using SCs. In this discussion there are several topics regarding the solutions developers have been using in their web applications to access SCs, but there is not a definitive in order to address this issue. The last comments in this discussion lead us to conclude that the Google Chrome developers are waiting for World Wide Web Consortium (W3C) to address this issue in a project that we will explain later in this section.

In healthcare industry there are several projects whose main goal is to develop a system where SC holding patient's records can communicate with web applications using web browsers. In [Chan et al., 2001], its authors used the *Java Card*[5] and Java Card Web Servlet (JCWS) technologies to deploy a SC applet which is stored within the card and can be loaded into the web browser. This is an attractive solution does not required to download and install new software. However, this is a specific solution to a specific problem that cannot be generally adopted to other types of SCs. It also requires the use of Java-compliant SC[6]. Java Card technology was invented in 1996, and it enables applications developed in the Java Card language (a subset of the Java programming language) to run on SCs [Sauveron, 2009]. In a similar work [Chan, 2000, 2003] an application was developed to provide access to SCs through an *HTTP* based interface protocol. This applications runs inside the SC, and can be seen as a web server which handles *HTTP* requests from the web browser. This solution shows low performance, because there are many operations executed inside the SC, which has low processing power.

The approach brought by [Starnberger et al., 2010] tries to standardize the access to cryptographic devices. For this purpose, the authors developed an application which plays the role of a proxy, enabling access from arbitrary web applications to arbitrary SCs. In this project they also defined strict policies regarding access to SCs, in order to protect them from malicious web applications. Once again users must install third-party software to perform the

[3] https://developer.mozilla.org/en/JavaScript_crypto#Signing_text
[4] http://code.google.com/p/chromium/issues/detail?id=73226
[5] http://www.oracle.com/technetwork/java/javacard/overview/index.html
[6] http://www.oracle.com/technetwork/java/javacard/overview/index.html

desired cryptographic operations. Thus, not requiring a new piece of software for each web application.

The product *SConnect*[7] by *Gemalto*[8] and referenced in [Sachdeva et al., 2009, Lu et al., 2011] is similar to the work that we intended to develop in this dissertation. The solution they propose is also plugin-based, and web applications can access SCs through JavaScript (JS). Besides connectivity, security was also a major concern of its developers. For instance, a web application which wants to communicate with a SC must have a valid credential. This solution was integrated in a wider web framework targeted to governments called *Coesys eGo*[9]. The *SConnect* is a closed and paid solution, thus we are not able to test its features and its OS support, SCs, and web browsers.

Cardboss[10], a paid product of *Comet Way*[11], also seems to provide a plugin-based approach with a corresponding JS API. At this time, they are only providing versions of their plugin to Microsoft Windows OSs.

PKI-Facille[12] is another solution from industry developed by *SmartCon*[13], which also seems to follow an identical approach to *Cardboss* and *Sconnect*. The (lack of) information available in its homepage is not sufficient to conclude if it is free, and to fully understand its technical features.

In the field of *USB* Smart Cards there are some relevant projects intended to making SC truly portable among operating systems and web applications. [Lu and Ali, 2010, Lu et al., 2009] contributed with a framework in which users can use their *USB* Smart Cards through a web browser. This framework is composed of an application which is stored inside the card and loaded into the host operating system. The application is responsible for establishing the communications between users, web browsers and web applications. The authentication to web applications is the only operation provided by this framework. Regarding OS support, this framework was successfully tested in Microsoft Windows. If specific drivers — like *HID* and *MSD* — are installed on the OS, this framework can be easily used in GNU Linux and Mac OS X without additional software.

A number of online communities are also concerned with the current solutions. The W3C, developer of web standards, recently created the project *Web Identity Working Group Charter*[14]. One of the deliverables of this project is a Cryptographic API. However, it is not explicit if this API will offer support for SCs.

[7] http://www.sconnect.com/News/index.html
[8] http://www.gemalto.com/
[9] http://www.gemalto.com/public_sector/solutions/coesys_egov2_0_version3.html
[10] https://cardboss.cometway.com/
[11] http://www.cometway.com/
[12] http://www.smartcon.com.br/cms4/archives/16
[13] http://www.smartcon.com.br/
[14] http://www.w3.org/2011/08/webidentity-charter.html

2.2 Smart Card Access Libraries

Enabling smart card functionalities in applications can be achieved by several means. One can use the available and native cryptographic libraries in operating systems, making the application restricted to a specific host. Low level interfaces can be used to achieve interoperability among OSs, but they require an extensive knowledge of the SC middleware. Figure 2.1 has a generic description of how the different layers are connected among the OSs.

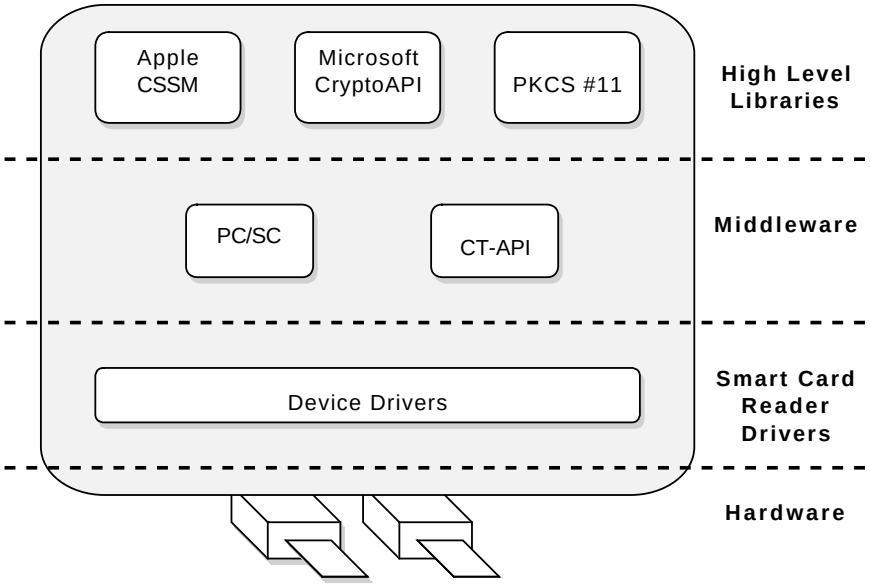


Figure 2.1: Availablele Libraries for SC Access

As shown in Figure 2.1, the drivers of the Smart Card Reader (SCR) are responsible for connecting SCs to OSs. It is expected that they conform to a common middleware standard, like *PC/SC*[15] or *CT-API*.[16] These standards facilitate the development of applications which support SCs, and the integration of SCs, SCRs, and OSs.

PC/SC is the de facto standard for smart card access and is available for several OSs: Microsoft Windows, Mac OS X and GNU Linux. This standard ensures that SC, SCRs, and computers made by different manufacturers work together. [Sachdeva et al., 2009] use this standard as the communication layer between their plugin and the SCs.

An application that uses *PC/SC* as the library for accessing SCs is more generic in practice. It will support all SCs which conform to this standard (most of them are), and it will not depend on the host specific libraries. However, the development is harder because it is a low

[15] http://www.pcscworkgroup.com/
[16] http://www.linuxnet.com/documentation/files/ctapi.html

level API. The communication between applications and SCs is possible using Application Protocol Data Units (APDUs), which can be seen as data packets which carry instructions or information, from or into the SC. Although this library gives the ability to communicate with several kinds of SCs, one must always know the behaviour, and the available functions and informations of each SC he specifically wishes to support.

Operating systems also offer support for SC access. There are cryptographic libraries in OSs which, among other features, offer dedicated functions to access SCs. These libraries provide a better abstraction of cryptographic functions and SCs, which make the development easier.

Microsoft OSs offer the *Cryptographic API (CryptoAPI)*[17] library. This library is designed to hide the details of cryptographic functionalities, providing applications with "pluggable" cryptography. For each SC there is a corresponding *Cryptographic Service Provider (CSP)* which does the mapping between cryptographic functions — exposed through CryptoAPI — and the low-level commands — accessible through the Win32 SC APIs[18].

In AppleOSs — *Mac OS X* and *iOS* — we can find dedicated libraries for cryptographic purposes. The first one of these libraries is *Cryptographic Services*[19], and it supplies the following features: encryption and decryption, key management, strong random number generation, secure communition using Secure Sockets Layer (SSL) and Transport Layer Security (TLS), and secure storage using Apple's specific features like *FileVault*[20] and *iOS File Protection*[21]. Another library is Common Security Services Manager (CSSM)[22], which is Apple's implementation of Common Data Security Architecture (CDSA)[23]. Through this library it is possible to access SC related mechanisms. Starting on *Mac OS X v10.7*, CSSM is considered deprecated and it should only be used when standard *Cryptographic Services* do not supply the desired features.

The Public-Key Cryptography Standards #11 (PKCS #11)[24] — also know as *Cryptoki* — developed by *RSA Laboratories*[25] is an API for cryptographic hardware access. Like *CryptoAPI* and CSSM, PKCS #11 also isolates applications from the cryptographic hardware. This de facto standard is available in *Mac OS X*, *Microsoft Windows* and all *GNU Linux* distributions, and it is supported by many SC vendors [Sachdeva et al., 2009, RSA Laboratories, 2004].

[17] http://msdn.microsoft.com/en-us/library/ms953432.aspx
[18] http://technet.microsoft.com/en-us/library/dd277376.aspx
[19] https://developer.apple.com/library/mac/#documentation/security/Conceptual/cryptoservices/Introduction/Introduction.html
[20] http://support.apple.com/kb/HT4790
[21] http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf
[22] https://developer.apple.com/library/mac/#documentation/security/Conceptual/cryptoservices/CDSA/CDSA.html
[23] http://www.opengroup.org/security/cdsa.htm
[24] http://www.rsa.com/rsalabs/node.asp?id=2133
[25] http://www.rsa.com/

The operating system specific libraries are the best option when one needs to deploy a native application, when compared to PKCS #11. They are updated more often — enhancing the overall system security — and they provide better application integration. [Sachdeva et al., 2009] However, such solution is not portable to different operating systems.

The Portuguese Citizenship Card (PCC) vendor provides an additional library to communicate with its SC, which is called *eID Lib API*, in addition to a PKCS #11 module. In this library we can find several methods to extract information from the SC, but it is not suited to perform cryptographic operations, such as the creation of DS. This module is available as a C++ dynamic library for all major OSs, but there are wrappers for *Java* and C# [Agência para a Modernização Administrativa, 2007].

2.2.1 A Short Introduction to PKCS #11

The PKCS #11 standard defines an API for SC interaction: from inspection operations to cryptographic functions, there are many methods developers can use to take full advantage of SCs. Usually, we can find implementations of this standard in *C*, but it can be used in C++ applications, and it is even possible to find wrappers for many other languages, *Java*[26] for example. The main advantages of PKCS #11 is the device-independence and object-oriented approach that isolate the development of applications from the details of the cryptographic devices.

In Figure 2.2 there is a general description of how *Cryptoki* manages to connect SCs to applications. *Cryptoki* provides an interface to the cryptographic devices attached to the computer, through the concept of "slots". A slot is a device that may contain a "token". A token is a cryptographic device— like a SC —that can be present in the slot. A convenient feature of PKCS #11 is the fact that software emulated tokens can be seen by applications as a regular physical token, due to the logical view *Cryptoki* provides of slots and tokens. Moreover, *Cryptoki* handles the connections from applications to SC, and if the library is initialized correctly, it can handle requests from a threaded application without problems.

From this point on, we must always recall the concepts of *slot* and *token*. A token represents a cryptographic device, and a slot represents a device where tokens are inserted, like a SCR.

Visibility

The access to objects and functions in the PKCS #11 API is restrained, because it depends on the permissions of the session that is established with the SC. We can define an established

[26] http://docs.oracle.com/javase/1.5.0/docs/guide/security/p11guide.html

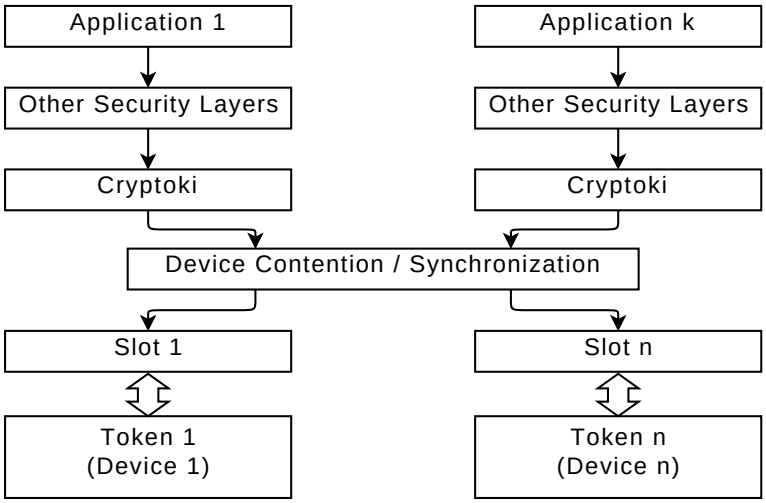


Figure 2.2: General Cryptoki Model. Adapted from [RSA Laboratories, 2004]

session as the moment when an application instructs the PKCS #11 library it she will start using SC features — such as the creation of DS — or access the stored data — as it is the case of data of private keys. Some permissions are strictly related to the type of SC user that is connected to the device. In *Cryptoki* the following users are available: *normal user* and *security officer user*. The PCC does not provides a Personal Identification Number (PIN) for a *security officer* user [Agência para a Modernização Administrativa, 2007]. The creation of a DS is, for instance, a function that requires a user to be logged in with the SC.

In Figures 2.3 and 2.4 we overview the states in read/only and read write/sessions. As shown in those figures, there are some functions that just authenticated users can perform.

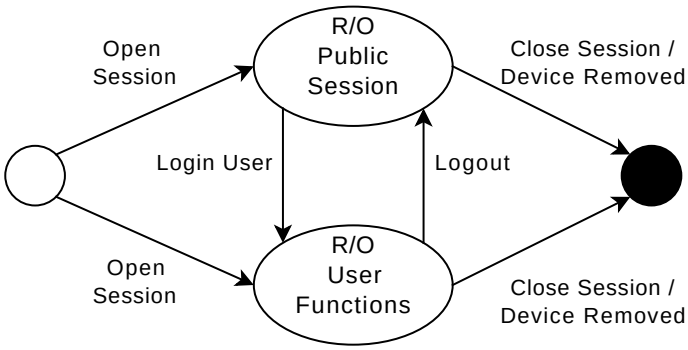


Figure 2.3: Read Only Session States in PKCS #11. Adapted from [RSA Laboratories, 2004]

PKCS #11 Objects

A PKCS #11 compliant SC can store several different kinds of information, like *Public Key Certificates* and informations of cryptographic mechanisms. In order to better understand

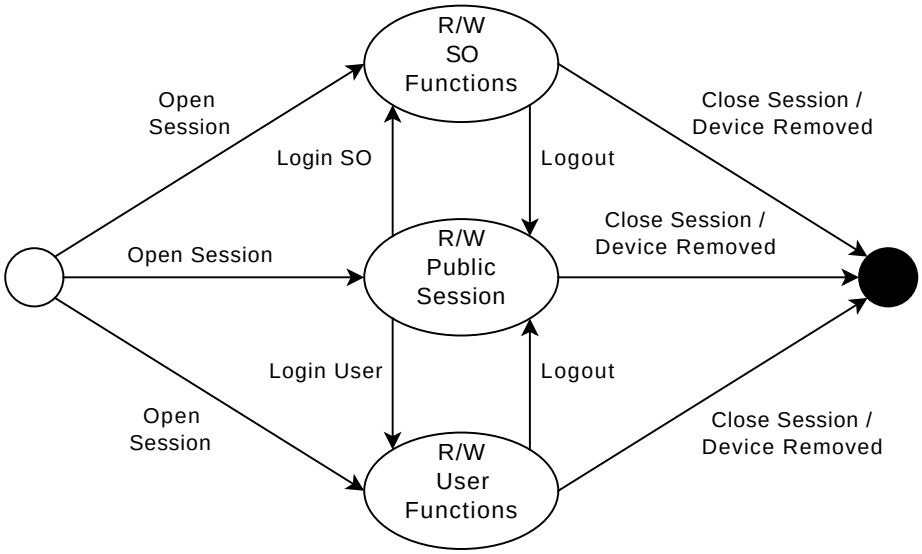


Figure 2.4: Read Write Session States in PKCS #11. Adapted from [RSA Laboratories, 2004]

the relation between objects and their type, the official PKCS #11 specification [RSA Laboratories, 2004] defines a hierarchy shown in Figure 2.5. In addition to storing values, objects can be used to perform operations, like in the creation of DS, where a reference to a private key must be specified in order for the SC to known what key to use.

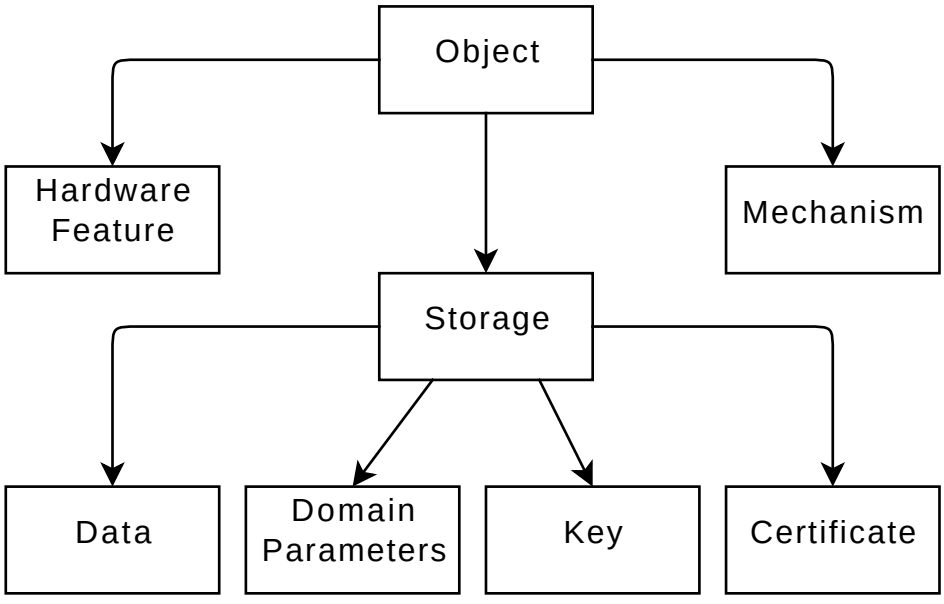


Figure 2.5: Objects Hierarchy in PKCS #11. Adapted from [RSA Laboratories, 2004]

As we can see from Figure 2.5, there are three main types of objects: *storage*, *hardware feature*, and *mechanism*. A *storage* object can be used to store keys (public keys, private keys, and secret keys), certificates (*X.509* Public Key Certificates, *WTLS* Public Key Certificates,

and *X.509* Attribute Certificates), and other informations like data and domain parameters. The other two main types hold informations regarding the physical characteristics of the SC, like user interface features (Hardware Feature), and supported cryptographic mechanism (Mechanism).

The objects in the PKCS #11 standard are composed by attributes which are responsible for storing meaningful values. An attribute has always a type, but it may not have its value defined, for instance: in an object holding a *X.509* Public Key Certificate we can find the attribute *CKA_Subject* which stores a Distinguished Encoding Rules (DER) encoded array of bytes containing the certificate subject. Besides storing data, attributes may also be used to differentiate objects: for instance, in a *hardware feature* object, the value of the attribute *CK_HD_FEATURE* influences the hardware feature (clock, monotonic counter, user interface) to which the object is referring to.

In order to use, inspect, or manipulate objects, the PKCS #11 standard offers several functions for that purpose. We can find dedicated functions to search for objects, get attributes from objects, and even manipulating them (create, copy, and modify). The process of finding objects requires that developers specify a template which matches the properties they want to find in that object. Example 2.1 shows a template for a private key object, with an id $0x45$.

```
1  //the object class that we want to find
2  CK_OBJECT_CLASS keyClass = CKO_PRIVATE_KEY;
3  //the key id
4  CK_BYTE         keyID    = 0x45;
5
6  //the template which represents a private key object
7  CK_ATTRIBUTE p11ClassTemplate[] =
8      {
9        { CKA_CLASS, &keyClass, sizeof(keyClass) },
10       { CKA_ID   , &keyID   , sizeof(CK_BYTE)  }
11     };
```

Example 2.1: Definition of a PKCS #11 Object Template

The PKCS #11 API is very flexible: whenever a developer wishes to access only a given part of an object he can specify exactly which attributes he wants to be retrieved from the SC. For that purpose, he must specify a template containing the attributes, like in Example 2.2, where it is defined a template to retrieve label, key type, id, start date, end date, and subject of what could be a *Public Key Certificate*.

In Section A.2 there is a full description for each one of the available objects in the PKCS #11 standard.

```
1 CK_ATTRIBUTE template[] = {
2     {CKA_LABEL      , NULL_PTR , 0 },
3     {CKA_KEY_TYPE   , NULL_PTR , 0 },
4     {CKA_ID         , NULL_PTR , 0 },
5     {CKA_START_DATE , NULL_PTR , 0 },
6     {CKA_END_DATE   , NULL_PTR , 0 },
7     {CKA_SUBJECT    , NULL_PTR , 0 }
8 };
```

Example 2.2: Definition of a PKCS #11 Attribute Template

PKCS #11 Functions

The PKCS #11 standard defines several categories of functions to inspect information in SCs and to instruct SCs to perform operations, like creating DSs and encrypting data. The categories we present next are some of the most important:

- **General Purpose -** The functions in this category are mainly used to initialize or finalize accesses to a PKCS #11 module.
- **Slot and Token Management -** The functions in this category have the goal to get information from the slots attached to the computer and from the tokens inserted in such slots.
- **Session Management -** The functions in this category are used to start or finalize connections with SCs.
- **Object Management-** The functions in this category can be used to search and search and get objects from SCs, as weel as getting informations from objects stored in SCs.
- **Message Digest -** The functions in this category are used to create digests of data.
- **Signing and MACing -** The functions in this category are mainly used to create DS of data.

Using a PKCS #11 module in any given application requires one to first initialize the library. In this initialization process, the PKCS #11 will allocate any needed resources and prepare the system for SC connections. In Example 2.3 we show the usual steps one must take to initialize a *Cryptoki* module. First, if the application performs multi-threaded access to the PKCS #11 library, the structure `CK_C_INITIALIZE_ARGS` must be initialized with pointers to functions for *mutex* management. Then, an entry point for the *Cryptoki* library must be obtained using `C_GetFunctionList`. Finally, it is time to initialize the PKCS #11 using `C_Initialize` and the structure `CK_C_INI_ARGS` as its parameter. If the application is single threaded, the parameter can be `NULL`. Once an application is done using the PKCS #11

```
1  CK_C_INITIALIZE_ARGS args;
2
3  args.CreateMutex   = (...) //pointer to a function that creates a mutex
4  args.DestroyMutex  = (...) //pointer to a function that destroys a mutex
5  args.LockMutex     = (...) //pointer to a function that locks a mutex
6  args.UnlockMutex   = (...) //pointer to a function that unlocks a mutex
7
8  //Loading all the functions from the library
9  rv = (*pC_GetFunctionList) (&pkcs11Functions);
10
11 //Initializing the pkcs11 Library
12 rv = (*pkcs11Functions->C_Initialize) (&args);
```

Example 2.3: Initializing the *Cryptoki*

module it should call the function `C_Finalize(NULL)` to close all connections to SCs and deallocate resources.

Since *Cryptoki* provides a high level abstraction of SCs, as well as their operations and data, the access to the information inside SCs always follows the same pattern. Typically, the first step in this process is to verify the number of available items present in the SC. In the following step it should be allocated enough space to accommodate the list of available items. Finally, it is possible to iterate through that list and access the information about each item. In Example 2.4 we show how to iterate through the information of all the tokens inserted in the computer. The first call to the function `C_GetSlotList` with the second parameter as NULL indicates that we want to be retrieved in `count` the number of available slots with a token present. This process can be applied to many different items, such like mechanisms and PKCS #11 objects —in these cases one must use the specific functions for mechanisms and PKCS #11 objects.

2.3 Developing a Web Browser Plugin

The development of a plugin is strictly tied to the web browser where it will be installed. Currently, we can differentiate Netscape-based web browsers from Internet Explorer. Among the first type of web browsers (i.e., Mozilla Firefox, Google Chrome, Safari, Opera) one can use Netscape Plugin Application Programming Interface (NPAPI)[27] as the development API. Intuitively we may think that, if one writes a plugin using NPAPI, it will run in all Netscape-based web browsers. However, if this plugin uses libraries specific to a given web browser, it will not be possible to integrate it with the others. The same reasoning can be applied to the relation between the plugin and the OSs, of course. One plugin that uses libraries specific from a OS is going to be platform-specific.

[27] https://developer.mozilla.org/en/Gecko_Plugin_API_Reference

```
1  CK_ULONG        count;
2  CK_SLOT_ID_PTR  pSlotList;
3  CK_RV              rv;
4  CK_TOKEN_INFO      info;
5
6  //getting the count of slots which have tokens
7  rv = (*pkcs11Functions->C_GetSlotList) (CK_TRUE, NULL, &count);
8  assert(rv==CKR_OK);
9
10 //creating enough space in order to store the list of slots
11 pSlotList = (CK_SLOT_ID_PTR) malloc(sizeof(CK_SLOT_ID) * count);
12
13 //getting the slot list
14 rv = (*pkcs11Functions->C_GetSlotList) (CK_TRUE, pSlotList, &count);
15 assert(rv==CKR_OK);
16
17 //getting the information from the Library
18 for(int i=0; i < count; i++)
19 {
20     rv = (*pkcs11Functions->C_GetTokenInfo) (pSlotList[i], &info);
21     assert(rv==CKR_OK);
22
23     processData(info);
24 }
```

Example 2.4: Getting Information about a Token in *Cryptoki*

Google Chrome provides also other APIs for plugin development like Native Client (NaCl)[28] and the Pepper Plugin API (PPAPI). NaCl provides a mechanism for safely execute platform-independent untrusted native code in a web browser. PPAPI is a branch of the NPAPI, which is stated by Google to address the portability and performance issues. Since the development of plugins using this mechanism is restricted to certain libraries provided by Google Chrome, it is possible to isolate malicious software from the rest of the system. Trusted code can perform privileged operations outside this mechanism, while untrusted code cannot. In Google Chrome, NPAPI plugins run outside of this mechanism[29]. This mechanism is called *Sandboxing* and it will be explained in Section 2.4.

Internet Explorer only supports Microsoft specific APIs, namelly *ActiveX Controls*[30]. An *ActiveX Control* can be seen as a library that can be used in Microsoft applications to enhance their base features. These controls have unrestricted access to the OS, and they can be developed in C, C++, and *Visual Basic*.

In Figure 2.6 we summarize the different types of plugin development interfaces that some web browsers support.

As we can see, developing a generic plugin that can be easily integrated with all web browsers is hard: it depends on the available plugin mechanisms and libraries in each web browser. For

[28] https://developers.google.com/native-client/

[29] http://www.chromium.org/nativeclient/getting-started/getting-started-background-and-basics

[30] http://msdn.microsoft.com/en-us/library/aa751968(v=vs.85).aspx

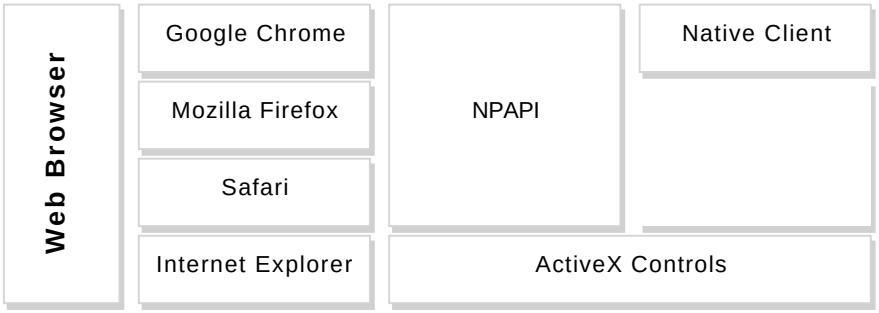


Figure 2.6: Available Techniques for Developing a Web Browser Plugin

that reason, several frameworks have been developed to ease the creation the web browser plugins:

- **FireBreath**[31] - can be used to create a web browser plugin that can run in several OS and has interfaces for the two main development APIs: NPAPI and *ActiveX Controls*. Relevant aspects: extensive documentation; the support for web browsers and OSs is well known; vast community; regular updates; good working examples[32]; with no costs.

- **Juce**[33] - is well suited for the development of software for different platforms, including web browser plugins. Relevant aspects: regular updates; cross platform and cross web browser support; good source code documentation but it lacks "getting started" guides; closed source applications require the payment of a fee;

- **Nixysa**[34] - can be used to generate source code for exposing plugin features to the NPAPI API. Relevant aspects: very poor documentation; the last release is relatively old (2009); and the cross platform support is not known; only supports NPAPI.

- **QtBrowserPlugin**[35] - is a solution for web browser plugin development. Relevant aspects: closed source applications require the payment of a fee; good documentation; cross browser and cross platform support.

2.4 Tampering Detection and Vulnerability Containment

Internet growth has helped software developers deploying applications more easily. Now, anyone can download an application directly from a software producer, and receive software

[31] http://www.firebreath.org/display/documentation/FireBreath+Home
[32] http://www.firebreath.org/display/documentation/FireBreath+Users
[33] http://www.rawmaterialsoftware.com/juce.php
[34] http://code.google.com/p/nixysa/
[35] http://doc.qt.digia.com/solutions/4/qtbrowserplugin/developingplugins.html

updates, for such application, whenever a new version is available. However, the safe delivery of software from Internet is in jeopardy, due to the spread of malicious code and the increasingly higher occurrences of phishing attacks that lure users to install fake software [Schiavo, 2010].

In order to minimize the risk of attacks, increase software security, and enhance user's confidence in applications, many techniques have been developed to protect both users and applications [Dasgupta et al., 2010]. Among these techniques, *Code Signing* and *Sandboxing* are being used in a wide variety of systems: from desktop applications to mobile OSs there are very well known usage examples of such methods, like *Apple's IOS* and *Windows* OSs.

In the following sections we will describe the background and the concepts behind *Code Signing* and *Sandboxing* techniques, and we will give some concise examples of their usage. Since we are developing a web browser plugin, we will also review the current support for these techniques among all major web browsers.

2.4.1 Code Signing

According to [Schiavo, 2010], *Code Signing* is an industry-recommended and widely-used defence against tampering, corruption and malicious infection. This technique can be used to enhance user's trust in the origin of a given software application, because users can verify precisely both the software's integrity and if it was developed by a known and trustworthy source.

The Public-key Cryptography (PKC) plays a major roll in the *Code Signing* technique, because it provides the means needed to prove the developer's identity and the integrity of software packages [Rubin and Jr., 1998]. In this process, a reputable Certificate Authority (CA) issues a *Public Key Certificate* for the software developer, who will use it to create a DS of the executable or script he wishes to deploy. Then, when a user fetches that software to his computer, he will check: (1) if the developer's *Public Key Certicate* can be validated by a trustworthy root CA, and (2) if the hash of the software package matches the DS sent by the developer. In case, the developer's identity is unknown or the software package is corrupted, the user is warned, and he can choose whether or not to proceed using the application.

Software developers can also use self-signed *Public Key Certificate* issued by a third-party CA. In such case, it may not be possible to users recognise the developer's identity as trustworthy, because none of the user's root CAs will validate the certificate of the developer as reliable. Still, software developers can publish on their official website the certificate chain which validates their identity, so users can validate successfully their identity and theirs software packages.

In Figure 2.7 there is a generic abstract representation of how *Code Signing* can be accomplished. In the first place, the software developer creates a DS from the software package he wants to publish, using his private key, and attaches the signature and his *Public Key Certificate* to the application — left side of the image. Then, users can download the software to their computers and check its real identity and integrity using the developer's *Public Key Certificate* — right of the image.

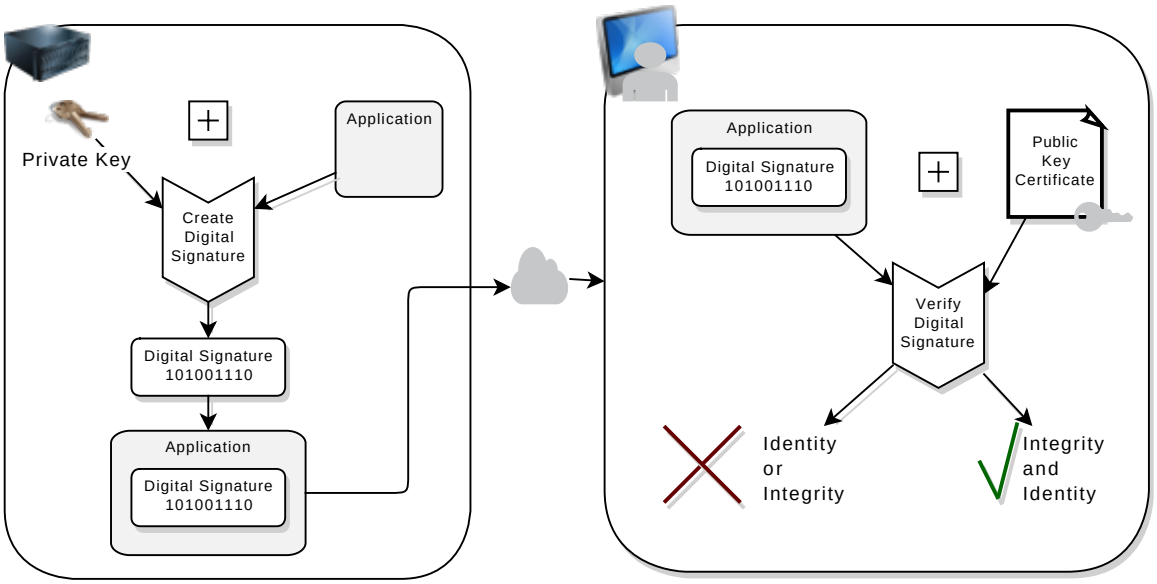


Figure 2.7: Abstract Representation of Code Signing

Usually, the above solutions use the Public-key Infrastructure (PKI) to achieve a trustworthy network among software developers and users, where reliable CAs issue *Public Key Certificates* to developers, which later can be recognised by one of the root CAs present in the users' computers. The GNU Privacy Guard (GPG)[36] defines a public exchangeable system of public keys that can be used to achieve the same goals as the standard PKI, therefore, it can be used in *Code Signing*.

Hashing software packages and publishing their result in a legitimate public platform may be considered another form of *Code Signing*. This solution can be found in many *open-source* and *free-software* projects, like the *Ubuntu*[37]. Although integrity can be verified, this solution does not provide any means to check identity.

As we can see, *Code Signing* promotes users' confidence in the origin of software packages, by defining restrict ways to check both identity of developers and integrity in any software package. Nevertheless, it fails when it comes to protect users against vulnerable software, and

[36] http://www.gnupg.org/
[37] https://help.ubuntu.com/community/UbuntuHashes

to impose accountability in software developers [Michener and Acar, 2000, Skoularidou and Spinellis, 2003]. *Code Signing* does not define any methods to check if a given application is bug-free or even if it will behave in a malicious manner. This is the reason why *Code Signing* must be used as a complement to other techniques such as *Sandboxing* in order to improve systems' security.

Operating Systems

At the present moment, we can find numerous implementations of *Code Signing* techniques among OSs, from desktop to mobile, and from paid to free of charge OSs. In fact, there are several operating systems which impose software developers to sign their applications using a valid *Public Key Certificate*. In the next paragraphs we will review the state-of-the-art among several OSs.

Apple The last software release of the Apple's desktop OS — Mac OS X 10.8 Mountain Lion — has a built-in mechanism which controls the origin of software sources named *Gatekeeper*[38]. By default, this mechanism only allows users to install software that was downloaded from the *App Store* or code-signed by a known trusted developer. However, it is possible to loose these restrictions and install software from unknown sources. In previous versions of the Apple's OSs, *Code Signing* was required to developers who wished to publish their software through the Apple's application store[39]. According to Apple's documentation[40], the user does not need to give additional permissions[41] to the application if the it has its code signed. Features like *Sandboxing* also depend on *Code Signing*. Developing applications for the mobile version of the Apple OS —the *iOS*— also requires a valid developer identifier, otherwise users won't be able to install applications. Unless they manage to get full root access to the device, to what is typically called *jailbreak*.

Microsoft Starting on the Microsoft *Windows XP*, a new technology called *Authenticode*[42] has been used to verify software sources, and also to digitally sign software packages. This technology can be used in many types of files, like executable (*EXE*), ActiveX controls, cabinet (*CAB*), and dynamic-link library (*DLL*)[43]. Usually, when a user downloads and tries to install a new application in his computer, this technology will check the software package integrity

[38] http://support.apple.com/kb/HT5290
[39] http://www.apple.com/osx/apps/app-store.html
[40] https://developer.apple.com/library/mac/#documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html
[41] http://developer.apple.com/library/mac/#technotes/tn2206/_index.html
[42] http://msdn.microsoft.com/en-us/library/ms537359%28v=vs.85%29.aspx
[43] http://msdn.microsoft.com/en-us/library/office/aa140234%28v=office.10%29.aspx

and identity, and warn him if any error occurred during that process. In this operation the user is asked if he wishes to continue using the software, even if the software origin is unknown. Microsoft also maintains a mobile operating system called *Windows Phone*, which has a native software package for legitimate application acquisition[44], where all published applications must be code-signed by a known developer.

GNU Linux Under the *GNU Linux* universe it is difficult to point out exactly the current state of the art for *Code Signing* for each one of the available distributions. For instance, there are distributions using *Code Signing* to protect users from malicious updates or patches. Debian[45]-based distributions use *Code Signing* for secure software distribution, where GPG is the auxiliary tool for such process [Dasgupta et al., 2010].

Android The *Android* OS maintained by Google is another good example of a mobile platform which demands *all* application developers to sign their software. By default, in an *Android* system all software must be installed through the *Google Play* store, where applications must be code-signed by software developers with a known and trustworthy certificate[46]. Still, it is possible to install software from unknown sources, using other locations than *Google Play*[47].

Java Despite the fact that the Java Virtual Machine (JVM) cannot be considered an OS, it is a virtualized system which can restrain which *Java* applications to run and which accesses to the real system can be performed. Thus, it is possible to use *Code Signing* to protect a *Java* application from tampering and to prove its source[48]. The JVM allows users to run applications without having their code signed, but once their signed the user must have the *Public Key Certificate* of the developer in his system.

Web Browsers

Extensions and plugins can enhance the user experience in web browsers, from block-advertising extensions, to video player plugins, there are several good examples that dramatically change the web browser standard functionalities. Even so, such third-party components can easily

[44] http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff402533%28v=vs.92%29.aspx
[45] http://wiki.debian.org/SecureApt#Secure_apt_groundwork:_checksums
[46] http://developer.android.com/tools/publishing/app-signing.html
[47] http://developer.android.com/tools/publishing/publishing_overview.html#unknown-sources
[48] http://docs.oracle.com/javase/tutorial/security/toolsign/index.html

obtain sensitive Information from its users, and jeopardize their security due to a defect on their code. This is why additional security mechanisms must be established in order to protect the users. The following paragraphs describe the current support for *Code Signing* in Google Chrome, Internet Explorer, and Mozilla Firefox.

Google Chrome The Google documentation for the Google Chrome web browser refers that it is possible to distribute extensions using the *CRX*[49] package format . This format defines how one developer may create a software package where he can put his *Public Key*, the extension signature, and the extension itself[50]. Google Chrome does not require to install extensions from known sources.

Internet Explorer Under Internet Explorer additional web browser core features can be expanded using *ActiveX Controls*. As mentioned before, can be code-signed and verified using the Microsoft *Authenticode* technology. Depending on the user security definitions of the web browser, *ActiveX Controls* without their code-signed or from unknown sources may not be able to run without user explicit consent.

Mozilla Firefox The current Mozilla documentation[51] states that anyone who is developing an extension or plugin can publish their software packges in a code-signed way. In Firefox there is also the possibility to develop code-signed JS scripts which can access expanded privileges[52]. These features do not restrict which software can be installed in the web browser, users can install whichever extension or plugin they wish, even they are from unknown sources, or are not code-signed.

2.4.2 Application Sandboxing

In the previous section we referred that *Code Signing* techniques do not protect from open vulnerabilities in an application, that can be exploited by malicious agents to gain access to users' personal data. That is the reason why *Code Signing* should be a complement to other techniques like *Application Sandboxing*. A system that implements a *Sandbox* where applications are ran, confines their execution in a such way that they cannot have greater privileges than they would have if running outside [Prevelakis and Spinellis, 2001], thus blocking

[49] http://developer.chrome.com/extensions/crx.html
[50] http://developer.chrome.com/extensions/packaging.html
[51] https://developer.mozilla.org/en-US/docs/Signing_an_extension
[52] http://www.mozilla.org/projects/security/components/signed-scripts.html

any operation that try to scale permissions, and thus restricting the damage a compromised application can cause [Goldberg et al., 1996].

Figure 2.8 depicts an abstract representation of a system that implements a *Sandbox*. Typically, an application that is developed to be confined within a *Sandbox* does not have direct access the real system. In return, it must use an API provided by the system that supplies an high level access to resources like file system or network. This middle layer between applications and the physical system is also in charge of controlling any harmful instruction, through a fine-grained analysis. It is usual in some *Sandboxes* implementations that developers have to define a list of permissions the application needs to have in order to execute, like in *Android*.

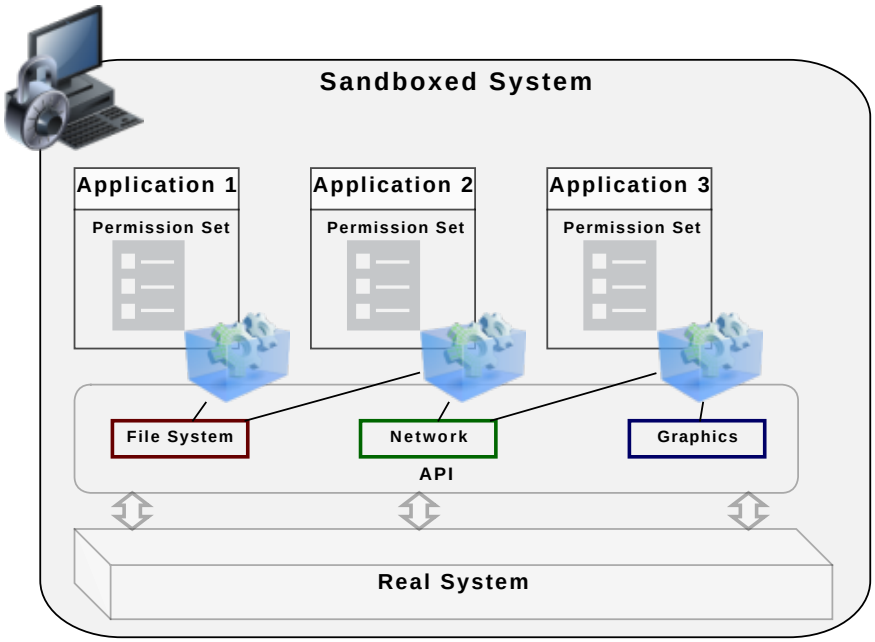


Figure 2.8: Representation of a System Using Sandboxing

The concept of *Sandboxing* applications is not new, in the version 4.2 of the *BSD* OS appeared a system call named *chroot*[53], whose purpose is to restrict the access of applications to a specific area of the file system.

According to [Prevelakis and Spinellis, 2001], the term *Sandbox* was first introduced by [Wahbe et al., 1993], where its authors developed a software approach to implementing fault isolation within a single address space. There are also other relevant projects which use this concept to enhance security on OSs, like [Prevelakis and Spinellis, 2001, Goldberg et al., 1996]. In the

[53] https://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man2/chroot.2.html

context of our project, the work [Goldberg et al., 1996] may be one of the most interesting ones, because they developed an user-level mechanism that monitors an untrusted application and disallows harmful system calls in order to protect users from attacks targeted to web browsers' helper applications.

An OS running in a virtual machine can be seen as a sandboxed system: the OS in the virtualized machine does not have access rights to the real physical hardware, and depending on the implementation of the virtual machine emulator, there are operations that can be restrained. Examples of virtual machine emulators are *KVM*[54], *VMware*[55], and *Oracle VM VirtualBox*[56], among many others.

Another good example of a system that confines the execution of the programs it runs is the JVM. Any *Java* application or applet are executed inside a virtual machine that can control the application permissions.

The concept of *Application Sandboxing* refers to any given system where applications run in a controlled environment, thus protecting users from compromised applications. Since this is a concept that can be applied for many systems, next we will review the current support for *Application Sandboxing* in OSs and web browsers.

Operating Systems

Modern OSs use sandboxing to protect users from ill-intentioned applications, from desktop to mobile operating system there are several usage examples of this trend. In the following paragraphs there is a review of the current native support for *Sandboxing* in OSs, we did not focus in other third-party implementations.

Apple Starting in *Mac OS X v10.5 Leopard*, Apple introduced a sandbox mechanism based on the *BSD* sandbox facility. Now a days, if a developer wishes to publish his application through the *Apple Store* and take advantage of features like *iCloud* and *Notification Center*, he must use the *Apple App Sandbox*[57]. According to the official documentation, the *App Sandbox* allows developers to describe which resources they want to use, to what they call

[54] http://www.linux-kvm.org/page/Main_Page
[55] http://www.vmware.com/virtualization
[56] https://www.virtualbox.org
[57] http://developer.apple.com/library/mac/#documentation/Security/Conceptual/AppSandboxDesignGuide/AboutAppSandbox/AboutAppSandbox.html

"entitlements"[58]. It is possible, however, to use applications that run outside the *App Sandbox*. The Apple's mobile OS *iOS* has also available a sandboxed, but in this case it is no possible to run applications outside its scope[59].

Microsoft In recent releases of Windows it is possible to install applications from an official Microsoft application called *Windows Store*[60]. Any released application through this store must declare the permissions it needs to execute, like access to a removable storage, to what Microsoft calls *App capability declaration*[61]. Installing applications outside of this store is also possible and does not need to perform such declarations. In *Windows Phone* there is a technology that restrains the *input/output* operations of applications named *Isolated Storage*[62]. According to the documentation of the *Windows Phone* all applications run in a sandboxed process[63].

GNU Linux According to [Dasgupta et al., 2010], *Ubuntu 9* uses the sandbox *AppArmor* to protect the system. Other techniques like *chroot jail* can be used to restrict the access to file system, thus creating a kind of a *Sandbox*. The SELinux[64] is a security enhancement that can be enabled in several GNU Linux distributions, such as: Fedora, Red Hat, Gentoo. This "enhancement" provides mechanisms to restrict the execution of programs, and the resources that each user is able to access. At this moment, we do not know any other implementations of sandboxes.

Android Applications for *Android* OSs are built using usually *Java*. Therefore, they run in a virtual machine which confines application execution. Each application must also have a set of permissions that define which system resources it will access. Any access to a resource that is not listed on the set of permissions is not permitted. The set of permissions is also used to alert *Android* users when they are about to install a new application, so they can decide to proceed with the installation.

[58] http://developer.apple.com/library/mac/#documentation/General/Conceptual/MOSXAppProgrammingGuide/Introduction/Introduction.html
[59] http://developer.apple.com/library/ios/#documentation/Security/Conceptual/Security_Overview/SecuritySvcs/SecuritySvcs.html
[60] http://www.windowsstore.com/
[61] http://msdn.microsoft.com/en-us/library/windows/apps/hh464936.aspx
[62] http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff402541%28v=vs.92%29.aspx
[63] http://msdn.microsoft.com/en-us/library/ff402533%28v=vs.92%29.aspx#bkmk_securityappsafeguards
[64] http://selinuxproject.org/page/Main_Page

Web Browsers

As mentioned in *Code Signing*, current web browsers features can be enhanced using extensions — which typically improve the user experience — and plugins — which usually improve the capabilities of web applications. In the folowing paragraphs we review the current support for *Sandboxing* for web browser plugins.

Google Chrome As referred in [Yee et al., 2009], Google Chrome has an integrated *Sandbox* that is used to confine the execution of a given type of plugin. The name of this technology is *Native Client* and it is intended to prevent side effects in the execution of plugins. At this moment, only plugins built using the PPAPI API run in this *Sandbox*. NPAPI plugins run outside, thus in a free and no confined environment. For those who are concerned with *Flash*, the Google Chrome version of the *flash player plugin* runs in *Native Client*[65].

Internet Explorer The Internet Explorer does not confine the execution of *ActiveX Controls*, but depending on the active security policy it may prevent from executing in the system.

Mozilla Firefox At this moment, all plugins run without restrictions in Firefox. We are not aware of any method to confine plugin execution in this web browser.

2.5 Summary

As shown in this chapter, there are several projects which purpose is to solve the lack accessibility to SCs from web applications. Some of them propose solutions that are not generic [Sauveron, 2009], and other ones are paid (for instance *Cardboss*) and do not provide an insight of their features. This is an interesting field where we could propose a new mechanism to create such accessibility, in an uniform and clean way between web browsers.

The connection between SCs and applications can be achieved using different kinds of libraries. As we described before, that are several libraries from low-level to high level complexity that can be used to get SCs and applications working together.

Creating a plugin requires building a software package which conforms to an API that is recognised by the web browser. As mentioned in this chapter, there are two main categories of APIs to build web plugins: NPAPI and *ActiveX Controls*. In order to abstract the plugin

[65] http://blog.chromium.org/2012/08/the-road-to-safer-more-stable-and.html

development process from the details of such technologies and to focus developers' work in the features they want to be available in plugins, many frameworks have been being deployed. These plugin development frameworks hide from developers the complexity of web browsers libraries, and sometimes offer cross platform support.

Finally, in this chapter we also reviewed to state of the art regarding techniques to protect users and applications: *Code Signing* and *Application Sandboxing*. These techniques can give us the possibility to protect our plugin and their users from the distribution of tampered or fake versions of it (*Code Signing*), and to protect users from the execution of malicious instructions achieved through attacks to an open vulnerability in the plugin (*Application Sandboxing*). Therefore, we presented the current support in OSs and web browsers for these techniques, which made us conclude that it is possible to use *Code Signing*, but *Application Sandboxing* is only possible in Google Chrome, at the present moment.

Chapter 3

Plugin Development

So far, we reviewed the current solutions for the problem we purpose to tackle, and the available technologies which might help us developing a web browser plugin to expose Smart Cards (SCs) to web applications through a JavaScript (JS) Application Programming Interface (API).

As we saw in Section 2.2, there are several available libraries that can be used to create connections between applications and SCs. In Subsection 3.1.1 we discuss the available technologies for SC access.

In Section 2.3 we found out that developing a plugin for a web browser requires one to develop a software package which respects to each web browser-specific API. At the present moment, there are two major libraries: the Netscape Plugin Application Programming Interface (NPAPI) that can be used in all major web browsers besides Internet Explorer; and the *ActiveX Controls* that is restricted to Internet Explorer. In Subsection 3.1.2 we present the strategy that we chose to develop our plugin.

Throughout this chapter we describe how we managed to successfully develop our plugin using some of the concepts we learnt reviewing the state of the art. In the second section we discuss and present the JS interface that will be available to web applications. Then, we give a little glimpse of the *Firebreath* framework and how it can be used to accomplish our goals. Some implementation details are also present in this chapter, where we expose the source code structure of the files and classes of the project. At last, we show how to properly use the plugin with a common usage scenario, and we present the several platforms where the plugin was tested.

3.1 Designing the Solution

The goal of this section is to discuss the most relevant topics of the development of the mechanism we propose.

The first, and one of the most important choices we had to make in our work, was the adoption of a library that would expose SCs to our software. In Subsection 3.1.1 we discuss the available options for SC access.

Secondly, we had to decide which technology would be more suitable for plugin development. In Subsection 3.1.2 we discuss the available mechanisms for the development of web browser plugins, and we present our choice.

Finally, in Subsection 3.1.3 there is an open discussion about several implementation topics.

3.1.1 Smart Card Access

At the present moment there are several technologies which enable cryptographic device in applications. As shown in Section 2.2, we can find several libraries that help developers to take full advantage of SC capabilities, from operating system (OS) specific APIs to open standards available in all major OSs. In the following paragraphs we present the reasons that led us to choose Public-Key Cryptography Standards #11 (PKCS #11), and why we discarded other libraries that would also allow us access SCs.

The OSs from Apple and Microsoft both have dedicated libraries for cryptographic purposes. It is through these libraries that application developers can perform operations, like encryption an decryption, and have their software working together with SCs. The main advantages of OSs specific libraries are the low complexity implementation details — they provide a high level abstraction of the hardware — and the update frequency — usually they are updated more often, which can increase their security and stability. These libraries are well suited to OS dependent applications, therefore they are not appropriate to our mechanism, where our effort is to create a uniform mechanism that exposes SCs to web applications. In a such situation it would require to have different implementations if we would desire to support several OSs.

A viable alternative to OSs dependent libraries if one wishes to have a cross platform support is *PC/SC*. As we discuss earlier in Section 2.2, this library is available in all major SCs and it is compatible with several SCs. Although *PC/SC* offers cross platform support, it is a low level library. Thus, one must know the implementation of the SC he wants to access, in order to take advantage of its features and contents.

The Portuguese Government offers a software library called *eID Lib API* that can be used to access the Portuguese Citizenship Card (PCC), and it is available for all major OSs. All of its features are dedicated to extract the information that is stored inside the SC, like address, age, or name. However, it does not supply cryptographic methods to be executed in SCs. Since one of our goals is to develop a mechanism that is able to create Digital Signature (DS), this library is not suited for our purposes.

Among all the available libraries we discussed, the PKCS #11 standard was the one that attracted us the most. This standard defines a high level abstraction of SC, and it provides several functions to inspect the contents of SCs and to perform cryptographic operations like the creation of DSs using SCs. Furthermore, it is possible to use PKCS #11 in all major OSs, because its features are not specific to any OS. As we already mentioned, there are available implementations of this library for all major OSs. The open-source project *OpenSC* provides an implementation of the PKCS #11 and it offers support for GNU Linux, Mac OS X, and Microsoft Windows. The PCC — our usage example — is compatible with this standard, and its vendor provides one PKCS #11 module for each on the major OSs. Due to these reasons, the PKCS #11 shows being an appropriate library for applications that are expected to be used in several OS.

Although MULTICERT actually required the adoption of PKCS #11 for this project, we would have chosen this standard anyhow, because of its cross-OS availability, its features and abstraction from the low level SC details. The PKCS #11 API will allow us to develop a plugin that does not depend on the OS specific features to access SCs, and we can ensure a better integration with our proof of concept. The adoption of this standard would also enable our plugin to be compatible with other SCs besides the PCC, because it is an open standard to which several SCs are compliant.

3.1.2 Plugin Development

As we discussed in Section 2.3, there are several methods to develop a web browser plugin. Typically, web browsers supply interfaces that developers can use in order to enhance their core functionalities. In this discussion we found that developing a web browser plugin would follow one of two general solutions: by the direct use of the web browsers interfaces, or by the means of a *framework*. Next we explain why chose a framework instead of the usual web browsers APIs, and we also described why we adopted *FireBreath* among the others.

In the first place, we had to decide if we would develop our plugin directly using the available API in web browsers. As we already mentioned, there two major APIs for the development of web browser plugins: the *ActiveX Controls* and the NPAPI. The first one is supported in

several web browsers, like Firefox, Google Chrome, Safari, and the second one is specific to Microsoft Internet Explorer. According to our goals, the plugin that we proposed to developed must be available for Google Chrome and Internet Explorer. Thus, it must comply to the NPAPI and *ActiveX Controls* interfaces. It would also require us to study each one these interfaces in order to understand how we could develop a plugin that can work in those web browsers.

During the initial phase of the project we explored other alternatives to develop web browser plugins. In this step we found several frameworks that hide the details of the web browser interfaces for plugin development. Such technologies allow developers to focus solely on the features they want to make available on the plugin, and sometimes they can help creating a plugin that can be successfully used in several web browsers. Due to these features, we decided that using a framework would allow us to save the time needed to understand the NPAPI and *ActiveX Controls* technologies. Thus, it would enable us to develop NPAPI and *ActiveX* plugins with the same source code core for the features we designed to our mechanism.

As we saw on Section 2.3, there are four main frameworks that help developers build web browsers plugins. The first one that we discarded from this set of frameworks was *Nixysa*, because it only has support for NPAPI plugins, it has poor documentation, and the last release is relatively old(2009). The three remaining libraries are very similar, they all have support for the major web browsers and OSs, and good documentation. Among these options we chose *FireBreath* since it is restricted to the development of web browser plugins, it has good starting guides, and has no costs — even when the web browser plugin is supposed to have its source code closed. The *Juce* would also be a good choice for developing our plugin, but we only became aware of this framework in an advanced phase of this project.

A full description of the *FireBreath* framework can be found in Section 3.3.

3.1.3 Implementation

Up to this point we discussed the several alternatives for accessing SCs and develop web browser plugins. From that discussion we decided that PKCS #11 would give us a good abstraction and a powerful access to cryptographic devices. We also decided that a framework would give us the possibility to create plugins for several web browsers with the same source code for the core features. Now we are going to discuss the major topics we had to face during the implementation step.

Linking with the PKCS #11 module

Linking the web browser plugin to the PKCS #11 module is perhaps the most important decision we had to make. This module is responsible for enabling SC capabilities in the plugin, and it has a major influence on the plugin security.

In this decision we had to choose between linking the module either dynamically or statically. Being statically linked means that the plugin would have the PKCS #11 library attached to its binary. In this situation the PKCS #11 library does not need to be present in the OS. Dynamically linked means that the PKCS #11 module would only be attached to the plugin at runtime, and it must be present in the OS.

Regarding security, the statically linking option shows being more reliable. Mainly because a malicious agent cannot change the PKCS #11 module without changing the plugin binary, since the library is a part of the plugin. It is also easier to deploy a statically linked plugin, because the end user would not need to install additional software, namely a PKCS #11 module, to run the plugin.

In our project we decided to dynamically link the PKCS #11 library to our plugin, only for testing purposes. Specifically, we chose to load the PKCS #11 library at runtime, where the plugin waits for the web application developer to insert the location of the module. We are aware of the security restrictions of our choice. However, in this initial phase of the project we though that it would be easier to test the plugin with different PKCS #11 modules, and in different platforms, using dynamic linking. Another advantage of this choice is that interested users on our mechanism can test it with other PKCS #11 modules besides the ones we initially thought (*OpenSC* and the module of the PCC).

In future versions of this project the plugin can still be deployed using dynamic linking, but additional security mechanisms would be needed. It could be used code signing in order to web browsers and OSs check if the plugin was tampered. The plugin would be deployed with a PKCS #11 module (like *OpenSC*) and it would also check the integrity of the module using *Code Signing*. It would also be interesting if web application developers could instruct the plugin to load a code signed PKCS #11 module. In this situation it would be needed: the PKCS #11 library, the Public Key Certificate used to sign the library, and the DS of the library. Using these elements the plugin can check if the module was not tampered (using the DS) and it can verify if it was developed by a known and trustworthy entity (using the Public Key Certificate).

Access to PKCS #11 objects

The PKCS #11 standard defines a hierarchy of objects, where each one has several attributes. As we presented on Subsection 2.2.1, the access to the objects — and its attributes — stored in SCs is performed through templates. One must specify a model of a template that matches the object he is looking for, in order to get a reference to that object. Once he has got the object reference, he must specify a model of an attribute template, in order to get the values from the object he is searching. From this small description we can conclude that the operation of searching for objects and its values in PKCS #11 is very verbose. So, we decided to wrap these details from the web applications, by supplying dedicated functions for each one of the PKCS #11 objects. The main reason that led us chose for this mode of operation is simplicity, we can expose the same amount and quality of information with less complexity and source code.

Returning the attributes of each PKCS #11 object

As shown in Appendix A.2, objects in the PKCS #11 standard can have several attributes. This is maybe one of the reasons why the access to the attributes of objects is performed using templates, so developers can access only to the information they want. However, in this project we decided to return all the attributes of each object.

We suspect that web application developers may not always want to get all the available attributes of a given object. We chose this operation mode only for testing purposes. In future versions of these plugin it would be interesting if each of the access methods to objects would have an argument containing a template. This argument could be an array of attributes defining the values that the developer wants to be delivered.

Handling binary data

The content of some attributes and the result of some operations in PKCS #11 is exposed using binary data. The result of a digest or the subject of a *Public X.509 Public Key Certificate* are a examples of data that is returned in the binary format. In the development of our plugin we decided that such data must be encoded in a language-independent format, because JS does not have a primitive way to represent binary data.

We could also encoded such data as strings, but it would produce unexpected results, because web browsers assume strings to be in the *UTF8* format[1].

[1] http://www.firebreath.org/display/documentation/Supported+JSAPI+types

So, we decided that the best solution for the representation of binary data would be encoding it in the *Base 64* format. Other formats could be adopted, like *hexadecimal*. We adopted *Base 64* because it is more efficient than hexadecimal.

Slot events

As referred in the goals of this project, warning we applications whenever a token is inserted or removed from a slot is a major feature that we wanted in our plugin. In order to accomplish this goal we used Document Object Model (DOM)-like events.

An alternative to our solution would be using JS callbacks registered on the JS interface of the plugin. These two approaches are very similar, but the first one is closer to the typical use of events in the development of web applications, where developers use DOM events to get information when for instance a user clicks an element of a web page.

A third option to expose these events to web applications would be through methods. In this situation a web application would invoke a method from the JS API of the plugin and it would block until an event occurs in a slot. This is a synchronous solution, and is not the most suited one, because it blocks the ordinary execution flow of web applications.

Since we decided to notify web applications of slot events in an asynchronous way, we had to make another decision. This time we had to decided if the plugin would be always watching for slot events, or if the web application developer can instruct it to start and stop listening for such events. Between these two approaches we chose the second one due to efficiency reasons, as an additional thread is needed in order to check whenever a token is inserted or removed.

Creation of Digital Signatures

In the PKCS #11 standard the creation of DSs using a SC needs developers to: (1) login into the SC, (2) feed the signature mechanism with data, (3) logout from SC.

The first and third steps are the most crucial ones, because: in the first place a Personal Identification Number (PIN) is needed in order to login; secondly, if the connection with the SC is not closed one may take advantage of this situation to create DSs without user's consent.

So, the first decision we made was to have total control over the connections to the SCs, thus the plugin must be responsible for prompting the PIN to users, and login and logout the user

into/from the SC. This is the only way that we can ensure that neither the web application learns the PIN of the user (when compared to the situation where the web application is in charge of this operation), and the connection with the SC is open only during the time needed to create the DS. Based on these decisions we decided to expose in the JS interface of the plugin two dedicated functions for digital signing purposes, one specific for the creation of DS from files, and another one from binary data. With this functions it is possible to specify if the plugin should ask for a PIN, and the data that is supposed to be signed. This solution give us the ability to control: (1) PIN prompting; (2) the precise time needed to create the signature, we now exactly when to login and logout; (3) the data that is supposed to be signed; (4) the time that the PIN is stored in the computer's memory, once the user is logged in we can erase the PIN from the memory.

3.2 API Design

In the development of the mechanism that we propose to enable SC capabilities in web applications — in a clean and uniform way among web browsers — we tried to create a consistent, simple, and easy to use JS interface. In that sense, we designed a browser and platform independent JS API with three different kinds of members: methods, attributes, and events. Each one of these interfaces will help web applications to: perform operations using SCs — methods —, get extra information about the plugin or the SC — attributes —, or even get notifications when a given event occurs — events. In Figure 3.1 there is an overall JS interface description.

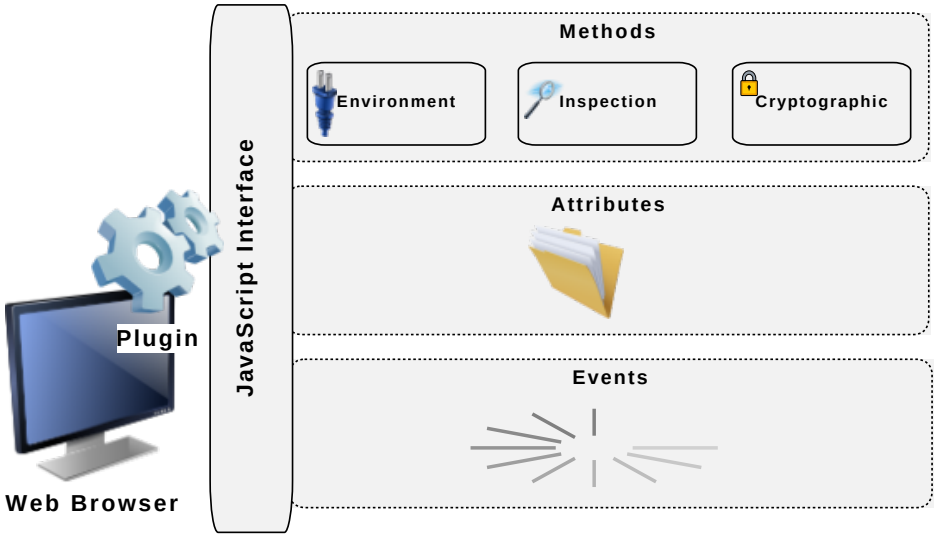


Figure 3.1: JS API Interface

In Appendix A.1 there is a complete API description of the JS interface, where we describe each method, attribute, and event.

3.2.1 Methods

SC-related features will be available to web applications as methods in the JS API, which will allow them to get information about SCs, and to accomplish cryptographic operations such as DS.

All the available methods in the JS interface of the plugin share the following three characteristics:

- *Fall into one category* - There are three types of methods, each type with distinct functions: one to setup the some plugin properties, one to inspect SCs features, and another one to perform cryptographic operations. These categories will be explained later.

- *Token-oriented* - Being token-oriented means that any operation will be performed with the concern to a given SC. At API level a SC is referred as a *token*.

- *Possibility to throw exceptions* - Since it is not always possible to execute successfully a method — due either to a wrong input, or an internal error in the function responsible for such method —, exceptions give the power to notify web applications whenever such situations occur.

Next, we will describe each method category.

Environment

This category of methods can be used by web applications to setup the environment where the plugin is running. There are two reason for the existence of this category of methods. The first reason is related to the PKCS #11 library, every time anyone wishes to use it he must first initialize it before using any of its functions. Once he finishes using it he must expressly instruct the PKCS #11 library to finalize, so all existing connections to SCs are closed, and resources freed up. Secondly, not all web applications would desire to receive warnings anytime a SC is either inserted or removed into/from the Smart Card Reader (SCR). So, for this purpose there are two methods which tell the plugin whether to start or stop listening for events in the SCR.

Inspection

Inspection methods give web applications the chance to get additional information from the SC, like available tokens in the user OS, available private keys, or private key details. Such information can later be used in methods as the cryptographic ones, which require SC specific details, like the cryptographic mechanism, or even identifiers for private keys.

In order to get information of any given data — whether about a SC, a public key certificate, or even a mechanism — one must always do:

1. A search for the available kinds of items he wishes to get more information, such as SC/token, or a private key. Once the search is complete, the plugin returns a list of integers, where each element identifies solely one of such item. The name of these kind of methods follows the convention `getAvailableItems`, where `Items` is replaced by the object name to search.

2. An access to the information inherent to the item he is looking for. In return, the plugin delivers a list containing the data from such item. The naming of these methods is similar to `getItemInformation`.

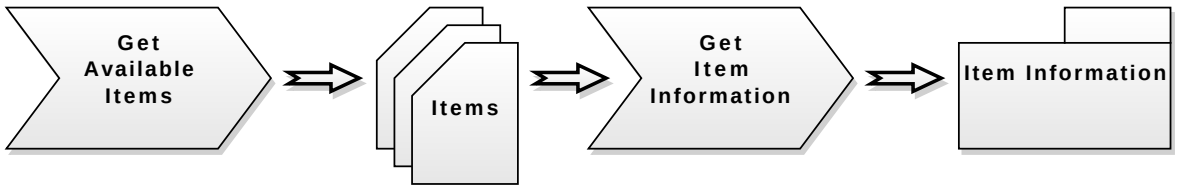


Figure 3.2: The *getAvailableItems* / *getItemInformation* Procedure

This scheme, as expressed in Figure 3.2, follows the PKCS #11 model for these kind of operations. An alternative to this scheme would be returning a map of item's identifiers to its informations. This solution could offer a lower performance, because additional information would have to be returned, even if not needed.

As we described in Subsection 2.2.1, the PKCS #11 standard defines an object model over the existing kinds of information that can be accessed. Such as: mechanisms, private keys, hardware features. In this scenario, if one wishes to get information about such items, he must use two different functions: one to get a reference to the object and another one to get values from it. In our work we decided to simplify this task, such as so we created an inspection method for each one of these objects, such as: `getPrivateKeyInfo` for private keys, and `getX509PublicKeyCertificateInfo` for *X.509* Public Key Certificate.

The information attached to each item available in the PKCS #11 library can be very extensive, there are items with more than ten fields. Due to this reason, the PKCS #11 standard

[RSA Laboratories, 2004] defines a template model, in which programmers can limit the fields they wish to inspect. For simplicity sake, we decided to retrieve a list of all the fields attached to each item. In contrast, we could have used a similar model to the one used by the PKCS #11 standard, where web application developers could define a list of which fields they want to get access.

Cryptographic

The last category of methods, but no less important, are the cryptographic ones. These methods let web applications use the cryptographic features available on SCs.

Currently, the plugin offers two kinds of cryptographic methods: one to create DS, and another one to create digests. While DS can be used by web applications to check its user's identity — in case of authentication — , or to help its users digitally signing a document — which has the same validity as its physical equivalent. Digests give the possibility to check a message integrity, whenever ah *hash* function is not available.

Both kinds of methods — DS and digest creation — have support for processing either files or bytes of data.

The PKCS #11 standard defines several functions for cryptographic functions, such as: encryption and decryption, Message Authentication Code (MAC)ing, and random number generation. However, the PCC — our case study — only offers functions for the creation of DSs and digests [Agência para a Modernização Administrativa, 2007]. This is why we chose not to implement any method regarding the remaining cryptographic functions defined in the PKCS #11 standard.

3.2.2 Attributes

The use of attributes in the JS interface of the plugin can be very useful to expose values, which do not need to be processed before returning their values to web applications.

Thus, we created three sets of read-only attributes, which web applications can use to get additional information about the plugin, SCs, and cryptographic mechanisms. The presence of such attribute values in the JS interface eases the work of web application developers, thereby avoiding documentation checks, and also helping minimizing copy&past related errors. Defining the attributes as read-only allowed us to expose values as typical *constant values* used in languages like C++. In the next paragraphs there is a description of each set of attributes.

Token-related attributes The PKCS #11 defines in its standard a field which holds the active flags for a given SC token, such as the presence of a random number generator, or even if the token is write-protected. Therefore, we decided to expose such flags, so web applications can have check SC features more easily.

Mechanism-related attributes Each PKCS #11 mechanism has a field to store its characteristics, like if it can be used in encryption or signing functions. The presence of these flags help web applications find out the best mechanism to use when they wish to create a DS or a digest.

Plugin-related attributes The intent of these set of attributes is to expose the following plugin properties: its version, string delimiters, and slot event types.

An alternative way to achieve the same goal without using attributes would be with methods, where each property would have a getter to its value. This solution would increase the plugin programming complexity, because it needs more lines of source code to produce the same effect as attributes.

3.2.3 Events

The interface events allow us to notify web applications about events, just like the ones used in DOM. In our plugin we used this interface to a callback anytime the state of the SCR changes, due to either an insertion or removal of a SC token. When this event is fired a callback in the JS is called, where the token and the event type (insertion / removal) are identified.

Instead of using events, we could have used directly JS callbacks, where web applications would register in the plugin a JS method to listen for occurrences in the SCR.

3.3 The Firebreath Framework

Firebreath is a lightweight, but nonetheless powerful framework which enables developers to build plugins that support all major web browsers and OSs. In this project we used the last stable release of the *Firebreath* framework — specifically 1.6 — and it supports the following web browsers and OSs:

- Windows

  - Internet Explorer 6 and later
  - Mozilla Firefox 3.0 and later
  - Google Chrome 2 and later
  - Apple Safari
  - Opera
- Mac OS X
  - Mozilla Firefox 3.0 and later
  - Google Chrome
  - Apple Safari 4 and later
- GNU Linux
  - Mozilla Firefox 3.0 and later
  - Google Chrome

In this section we will overview the *Firebreath* framework. First we will enumerate its requirements in order to have a fully operational system where *Firebreath* can be used, Subsection 3.3.1. Then we will describe the several steps needed to successfully compile and install a plugin using this framework, and how we managed to automate this tasks, Subsection 3.3.2. Finally, we will show how the framework can be used to create features that will be available to web applications, Subsection 3.3.3.

3.3.1 Requirements

The first thing that we must recall about *Firebreath* is that it is a C++ framework only; and it does not supply any means to compile, install or test plugins. Therefore, we will need additional development software like interpreters, compilers, build systems, or even Integrated Development Environments (IDEs), depending on the system we will compile the plugin.

In all OSs supported by *Firebreath* — *GNU Linux*, *Mac OS X*, and *Windows* — *Python*[2] must be installed at least at the plugin creation time. *Python* is used to run a script which creates

[2] http://www.python.org

the base source code structure of the plugin. According to the *Firebreath* documentation[3], it is recommended to use one of the following *Python* versions: `2.5`, `2.6`, or `2.7`.

The *CMake*[4] is the build system used by *Firebreath* to structure the plugin compiling definitions. The recommended *CMake* version needed to compile the plugin differs among OSs: `2.8` in *GNU Linux*, `2.8.8` in *Mac OS X*, and `2.8.7` in *Windows*.

Under a *GNU Linux* operating system the following software packages must be installed in order to compile the plugin:

- **GTK development libraries version 2.0** These libraries are used for drawing support
- **GNU Make**[5]**, GCC**[6] These tools are used to compile the plugin

Under *Mac OS X* operating system the following software packages must be installed in order to compile the plugin:

- **Apple's XCode**[7]
- **XCode's command line tools**

Under *Windows* operating system the **Microsoft Visual Studio**[8] IDE must be installed in order to compile the plugin.

The *Firebreath* framework must also be present in the system, it can be downloaded from its official download page[9].

3.3.2 Development Life Cycle of a Firebreath Plugin

The development life cycle of a plugin using the *Firebreath* framework, from its creation to its installing, can be divided into four steps, as shown in Figure 3.3.

1. **Creation -** The goal of this step is to:
   - define the plugin properties, such as: name, *MIME* type, Company Name, Description, among many others;

[3] http://firebreath.com/display/documentation/Creating+a+New+Plugin+Project
[4] http://www.cmake.org
[5] http://www.gnu.org/software/make
[6] http://gcc.gnu.org
[7] https://developer.apple.com/xcode
[8] http://www.microsoft.com/visualstudio
[9] http://firebreath.com/display/documentation/Download

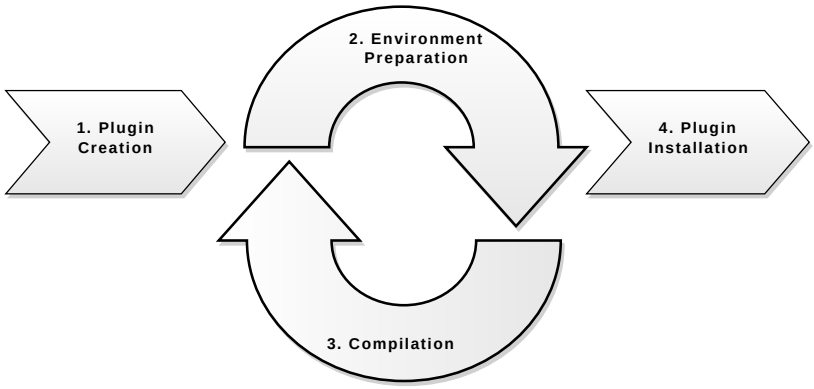


Figure 3.3: Firebreath Development Cycle

- generate the base source code structure of the plugin project, which later will be explained in Section 3.4.

In order to ease this task, the *Firebreath* framework offers a *Python* script — named *fbgen.py* — which automates the plugin generation.

Obviously, this step is only performed once in the development process of a plugin.

2. **Environment Preparation -** The goal of this step is to:

   - define the plugin project location and the build directory;
   - scan the plugin project for source files, so all dependencies are known at compile time;
   - find the plugin properties, such as additional libraries;
   - define the build type, it can be: *Debub*, *Release*, *MinSizeRel*, *RelWithDebInfo*; (this definition can be set later if the plugin is compiled using an IDE)
   - fetch external *Firebreath* libraries dependencies like *Boost*[10], if they are not present in the environment where the plugin will be compiled;
   - generate auxiliary build files, such as: IDE projects, and source files regarding the plugin core functionality.

   Once again, for this task *Firebreath* offers the following auxiliary scripts for each platform it supports, which will help creating a prepared environment for a successful compilation:

[10] http://www.boost.org

- *prep20xx.cmd* - a batch script for the *Windows* platform that generates a *Microsoft Visual Studio* compatible build setup;
- *prepcodeblocks.sh* - a batch script for *Unix* like platforms that generates a compatible setup for the IDE *Code::Blocks*[11];
- *prepeclipse.sh* - a batch script for the *Unix* like platforms that generates a compatible setup for the IDE *Eclipse*[12]
- *prepmac.sh* - a batch script for the *Mac OS X* platform that generates a *XCode* compatible build setup;
- *prepmake.sh* - a batch script for the *GNU Linux* platform that generates a compatible setup with *GNU Make* and *GCC*.

The created build environment should never be shared among systems, even among the same operating system, because all the build definitions — like system paths — are created according to the system they were created.

This step must be executed when someone is about to compile the plugin project for the first time, and must be repeated every time someone changes the plugin properties or definitions, or adds a new source file to the plugin project.

3. **Compilation -** The compilation step has the purpose of creating a binary compatible with a NPAPI plugin or an *Activex control*, or even both, depending on the operating system it is performed. Next we present the tools for which *Firebreath* offers support for compilation:

   - **GNU Linux** - *GNU Make* build system, or *Code::Blocks* and *Eclipse* IDEs
   - **Mac OS X** - *XCode* IDE or the command line tool *xcodebuild*
   - **Windows** - *Microsoft Visual Studio*

   The plugin project definitions should never be changed inside IDEs, with the exception for the build type and the target architecture, when these definitions are available.

   The plugin binary is only compatible with the operating system where it was compiled.

   This step must be performed every time the source code is changed.

[11] http://www.codeblocks.org
[12] http://www.eclipse.org

4. **Installation -** The installation process intents to expose to web browsers - or even OSs — the newly compiled plugin, and it varies depending on the operating system:

   - **GNU Linux**
     There are two ways to successfully install the newly created NPAPI plugin, making it accessible to web browsers, it can be installed to every user present on the operating system, or simply just to a single one.

     – *Install for everyone* - copy the created shared object file — the plugin — to `/usr/lib/mozilla/plugins`

     – *Install for a single user* - copy the created shared object file — the plugin — to `<USER_HOME>/.mozilla/plugins`

   - **Mac OS X**
     The installation process under a *Mac OS X* operating system follows the same philosophy as the one for *GNU Linux*, but with different locations:

     – *Install for everyone* - copy the created shared object file — the plugin — to `/Library/Internet Plugins`

     – *Install for a single user* - copy the created shared object file — the plugin — to `<User_HOME>/Library/Internet Plugins`

   - **Windows**
     Under a *Microsoft Windows* operating system the plugin can only be installed for every user, using the command line tool *regsvr32* to create a new entry in the *registry*. When a web browser is started it will check the registries for plugins and it will load the shared object associated to each registry.

     The plugin binary works as both a NPAPI plugin and an *Activex Control*.

After the plugin installation, the web browser where it will be tested must be restarted, so it can reload the list of shared objects.

If the plugin is under development in *GNU Linux* or *Mac OS X* the *Firebreath* documentation recommends to place symbolic links to the shared object in the installation folders, instead of copying it directly to those places. Thus, every time the plugin binary is created, it is also updated.

A Bit of Automation

Firebreath offers several scripts to ease the development life cycle of a plugin. However, one must always recall which scripts to use, their location, and their parameters, which may jeopardize the plugin consistency among OSs, and slow down the build process.

In our work we created additional mechanisms to automate the development — from the environment preparation to the plugin installation —, so anyone who is developing just needs to know if he wants to prepare the compilation, compile the plugin, or install it. For this purpose we created a simple batch script for each platform — one compatible with the *Windows* OSs and another one compatible with *Unix* like systems — with identical operations, and functionalities.

3.3.3 Using the Firebreath Framework

One of the most interesting features of *Firebreath* is the ease of interacting with JS. Currently *Firebreath* exposes four basic types of interfaces to JS: methods, properties, attributes, and events. In order to create these interfaces, *Firebreath* offers a C++ class named *JSAPIAuto* that must be inherited by at least one of the plugin classes. The *JSAPIAuto* hides the details of exposing features to the JS layer, and simplifies the amount of code needed for type casting between JS and C++, and vice versa.

Next we will present each one of the interfaces, and we will explain how one can expose each one to web applications.

Methods

A method provides a useful way to enhance the JS capabilities of web applications: it gives access to procedures that are not available in the regular JS provided by web browsers.

Usually, a method can accept zero or more arguments and it can optionally return a value. Whenever a method does not explicitly returns a value, the web application will receive a `undefined` value, which is the ordinary behaviour of a typical JS function without a return value.

Creating a new method using the *Firebreath* framework — which will be available through JS to web applications — requires registering the method as one member of the JS interface, and defining its behaviour.

The first step to create a new method — which will be accessible from the JS interface — is to define its behaviour. This is identical to any other regular method definition in C++: first one must to create its signature inside the *class* in the *header* file, and then create it in the *source* file, as shown in Example 3.1 where it is defined a method named `add_internal` which belongs to the class `MyPluginAPI` and calculates the sum of two integers.

Header File *(.h)*

```
class MyPluginAPI : public FB::JSAPIAuto
{
    public:
       (...)
         int add_internal(int a, int b);
       (...)
};
```

Source File *(.cpp)*

```
int MyPluginAPI::add_internal(int a, int
    b)
{
    return a + b;
}
```

Example 3.1: How to define a new method in *Firebreath*. Adapted from the *Firebreath* documentation[13]

Once the method behaviour is defined it can be exposed to the JS interface using *Firebreath* provided functions to register methods in the plugin JS API. In Example 3.2 there is an instance of such registration, using the special functions: `registerMethod` and `make_method`. The first parameter of `registerMethod` defines the method's accessible name from JS, in this example it is `add`. The second parameter is a pointer to a function that will perform the conversion from the JS values to C++ compatible types, and check the argument count. Such function can be generated using `make_method`, which needs one pointer to the method that will handle the JS request, and another to the object where it belongs — in the example the method that will handle the request is `add_internal` and it belongs to the class `MyPluginAPI`. The registration must be placed inside the object constructor for the `JSAPIAuto` derived class.

```
1 MyPluginAPI::MyPluginAPI()
2 {
3     registerMethod("add", make_method(this, &MyPluginAPI::add_internal));
4 }
```

Example 3.2: How to register a new method in *Firebreath*. Adapted from the *Firebreath* documentation[14]

Even tough JS is a dynamic, weakly-typed language, *Firebreath* ensures strong dynamic typing from JS (input) to C++ values, for a great majority of types. *Firebreath* will always try to match the input values to the ones in the method definition, whenever this operation fails an exception will be thrown to the web application. At the current stable version of *Firebreath* the following types are supported:

- arithmetic, such as: `int`, `long`, `short`, `char`, `double`, and `size_t`;
- boolean;

- string;
- container types compatible with the Standard Template Library (STL)[15];
- JS objects, such as methods for callback.

Any of the above types can be used as an output to the JS interface, with the exception for the container types. In this case it is only possible to return lists — `std::vector` — or maps — `std::map<std::string,...>`.

In the development of our plugin we used these kind of methods to create the functions from the JS API that will perform operations like: creating DS, and check for SC, among many others.

Attributes

Attributes can be used to expose values to the JS interface of the plugin. One must use attributes whenever getting — or setting — a value needs no special logic to handle such request.

The creation of a new attribute is simple. *Firebreath* requires only a registration where it must be defined: the accessible name from the JS interface, the associated value, and optionally define if it is a read-only attribute. In Example 3.3 there is a registration of two attributes: a read-writable named `readWriteValue` with the default string value `a string value`; and another one named `readOnlyValue` with read-only permissions, that holds the value `another string value`.

```
1 MyPluginAPI::MyPluginAPI()
2 {
3     registerAttribute("readWriteValue", "a string value"));
4     registerAttribute("readOnlyValue",  "another string value", true);
5 }
```

Example 3.3: How to register a new attribute in *Firebreath*. Adapted from the *Firebreath* documentation[16]

Once again it is possible to use anyone of the types referred previously in the description of methods.

We used attributes to expose certain constants to the JS interface, that otherwise would require web applications to have firsthand knowledge of their values. Example of such constants are token and mechanism related flags defined by PKCS #11.

[15] http://www.cplusplus.com/reference/stl

Properties

The use of properties in the JS API enables web applications to get — and set — values from the plugin, in the same way one can access member variables of any given class. This interface may resemble attributes, but they have different goals. A property must be used when its content needs to be processed before setting or getting its value. A property can have either read-write permissions — web applications can read and change the property's content — or read-only permissions — it is not allowed to change the property's value.

The first step in the creation of a new property is to declare the variable which will hold the value that both web application and plugin will have access to. Then, it is time to create a two member functions, where one will act as a value getter, and another as a value setter — if it is a read-write property. Example 3.4 shows the creation of a new property, having its `string` value stored in the member variable `m_value`, with the getter and setter `get_value` and `set_value`, respectively.

Header File *(.h)*

```
class MyPluginAPI : public FB::JSAPIAuto
{
    public:
      (...)
      std::string get_value();
      void set_value(std::string& val);
      (...)
    protected:
      std::string m_value;
};
```

Source File *(.cpp)*

```
std::string MyPluginAPI::get_value()
{
    return m_value;
}

void MyPluginAPI::set_value(std::string& ↩
    val)
{
    this.m_value = val;
}
```

Example 3.4: How to define a new property in *Firebreath*. Adapted from the *Firebreath* documentation[17]

At this point the property is just like a regular member variable of any other class, and it cannot be accessed outside the plugin. Thus, we have to register this property in the plugin JS interface, in a way similar to the method registration. To this end, *Firebreath* offers two special functions for registering properties: `registerProperty` and `make_property`. While `registerProperty` is in charge of defining the accessible name from the JS interface, and the function that will perform the conversions from JS values to C++ types; `make_method` is used to generate the source code of a such conversion function. Example 3.5 shows how to register a read-only property, and a read-write property, with the names `readOnlyValue` and `readWriteValue`, respectively. The difference between these two properties is the presence of a setter in their registration — if the function `make_property` is not provided with a setter, then the property will be read-only value. The registration must be placed inside the object constructor for the `JSAPIAuto`-derived class, just like in a method registration.

Regarding the supported types for properties, it is possible to use any of the types referred in the description of methods.

```
1 MyPluginAPI::MyPluginAPI()
2 {
3     registerProperty("readOnlyValue",  make_property(this, &MyPluginAPI::
      get_readOnlyValue));
4     registerProperty("readWriteValue", make_property(this, &MyPluginAPI::
      get_readWriteValue, &MyPluginAPI::set_readWriteValue));
5 }
```

Example 3.5: How to register a new property in *Firebreath*. Adapted from the *Firebreath* documentation[18]

Events

Events give plugins the possibility to warn web applications every time a given occurrence happens, just like normal DOM events — such as `onload` and `onmousemove` — firing callbacks in the web application JS. Such mechanism allows web applications to notify their users or even to adapt their interface according to a given event.

In order to create a new event one must declare it — inside the `JSAPIAuto` derived class definition — using the *Firebreath* macro `FB_JSAPI_EVENT`, respecting the syntax in 3.6. The macro expects one to identity the following event properties: name — it must be entirely lower case due to browser differences —, argument count, and argument types. One must be use arguments in his events anytime additional knowledge about it is required.

```
FB_JSAPI_EVENT({name}, {arg count}, ({arg types}))
```

Example 3.6: Event creation syntax in *Firebreath*. Adapted from the *Firebreath* documentation[19]

With the intent to exemplify how such declaration can be successfully achieved, Example 3.7 has the registration of a new event named `event`, which has two arguments: one `integer` and a `string`.

```
1 class MyPluginAPI : public FB::JSAPIAuto
2 {
3     public:
4       (...)
5       FB_JSAPI_EVENT(event, 2, (int, const std::string&));
6       (...)
7 };
```

Example 3.7: How to create a new event in *Firebreath*. Adapted from the *Firebreath* documentation[20]

After declaring the new event, all member functions from the `JSAPIAuto` derived class can warn web applications about its existence by triggering a special function responsible for that event. At the time the event is declared — like in Example 3.7 — *Firebreath* automatically generates a new function to fire such event, which name follows the template

`fire_{event_name}`. In Example 3.8 there is a function named `fire_event` firing the event declared in 3.7, using the values `123` for the first integer argument, and `string` for the second string argument.

```
1 void MyPluginAPI::fireEvent()
2 {
3     fire_event(123,"string");
4 }
```

Example 3.8: How to fire an event in *Firebreath*. Adapted from the *Firebreath* documentation[21]

The plugin that we developed uses these kind of evens to warn web applications every time a token is either inserted or removed in/from the SC.

3.4 Implementation

The first step in the creation of the plugin was to generate the initial source code structure of files and classes using the auxiliary scripts supplied by the *Firebreath* framework, as described in Section 3.3. In this operation the files presented in Table 3.1 were generated. We decided to name our plugin as *Smart Cards Everywhere* since our effort is to create a uniform and browser-independent mechanisms that exposes SCs to web applications.

The files `SmartCardsEveryWhereAPI(cpp | h)` are some of the most important ones in our project. The behaviour of the class `SmartCardsEveryWhereAPI` is defined in these files. This class is responsible for handling all the requests made to the JS interface of the plugin. In this class we registered all the members of the JS interface (methods, attributes, and events), and their behaviour as well.

In Figure 3.4 we present the class diagram of our project. For simplicity sake we used *packages* to represent C++ *namespaces*. On the right side and inside a rectangle of this diagram there is a representation of the classes we developed in our project.

On the left side there are the two main classes inherited by the classes of our project: `SmartCardsEveryWhere` and `SmartCardsEveryWhereAPI`. During the development phase we decided to create a `namespace` named `utils` dedicated to hold auxiliary classes. Within this `namespace` we created the class `Utilities` that has several methods to: dynamic library management, type conversion, and *mutex* management.

The class `SlotEventListener` is used to wait for changes in slots attached to a computer. In order to perform this operation it runs a thread that is constantly checking the status of

File	Description
PluginConfig.cmake	In this file it is possible to define several informations about the plugin, such as: name, *MIME* type, Description, and Company name. It is also in this file were we can define if we want to link our plugin with any of the libraries supplied by *Firebreath*, like *OpenSSL*[22].
CMakeLists.txt	It is in this file where the configurations for the compilation are written. Any cross platform library, or file, must be specified in this file.
Factory.cpp	This file contains the class that is responsible for creating the main plugin object, and for initializing and finalizing the plugin.
SmartCardsEveryWhere (cpp | h)	These files have the definition of the class *SmartCardsEveryWhere*, which is the main entry point of the plugin.
SmartCardsEveryWhereAPI (cpp | h)	These files contain the definition of the class *SmartCardsEveryWhereAPI*, which will handle all the JS requests. From methods, to events, this class will process each request to the JS interface of the plugin.
Mac/projectDef.cmake	This file defines additional compile instructions specific to the Mac OS X OS. Specific libraries of this OS must be referred here.
Win/projectDef.cmake	This file defines additional compile instructions specific to the Windows OS. Specific libraries of this OS must be referred here.
X11/projectDef.cmake	This file defines additional compile instructions specific to GNU Linux OSs. Specific libraries of this OS must be referred here.

Table 3.1: Initial Source Code Structure. Adapted from the *Firebreath* documentation[23]

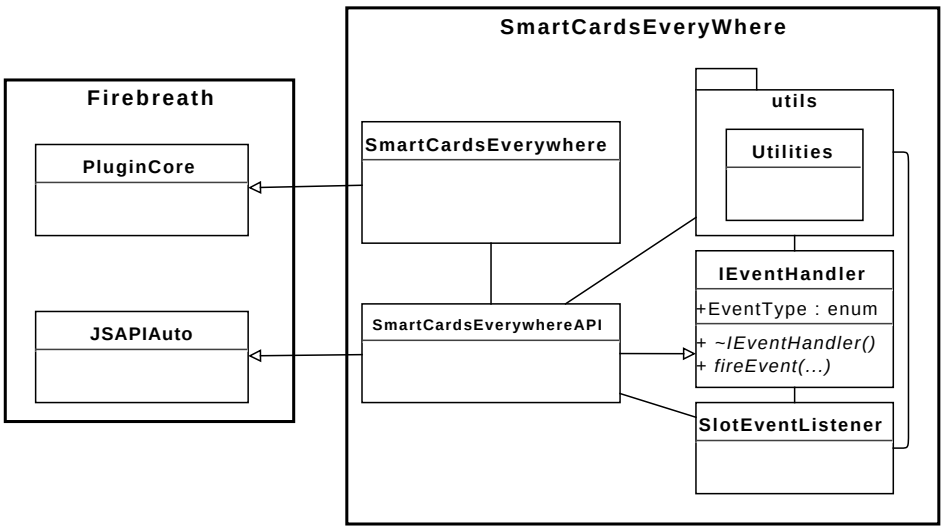


Figure 3.4: Class Diagram of the Plugin

the SCRs. The class `SmartCardsEveryWhereAPI` is notified when a change occurs, so it can warn web applications accordingly.

The communication between `SmartCardsEveryWhereAPI` and `SlotEventListener` is achieved through the class `IEventHandler`. This class defines an abstract behaviour of a

method that should be fired when a given event occurs. As shown in Example 3.9 the method is called `fireEvent` and it has three parameters: the first one identifies the function where the event occurred, the second indicates the event type, and the third the data associated to that event. Since `SmartCardsEveryWhereAPI` inheres `IEventHandler`, then it must implement this method. When a request is made to the JS interface for slot events, the class `SlotEventListener` is instantiated and a reference of `SmartCardsEveryWhereAPI` is passed.

```
(...)
enum EventType
{
    CHECK_RETURN,
    TOKEN_INSERTED,
    TOKEN_REMOVED
};
(...)
virtual void fireEvent(Utilities::FunctionsList funID, EventType evtID, void * data) = 0;
```

Example 3.9: Excerpt of the class IEventHandler

In Figure 3.5 there is a representation of the currently file structure of the project. Inside the folder `include` we can find the several external headers needed to compile our plugin. At this phase of the project these headers are all related to the PKCS #11 library. The file `Configurations.h` is used to define which headers to included depending on the OS where the plugin is being compiled.

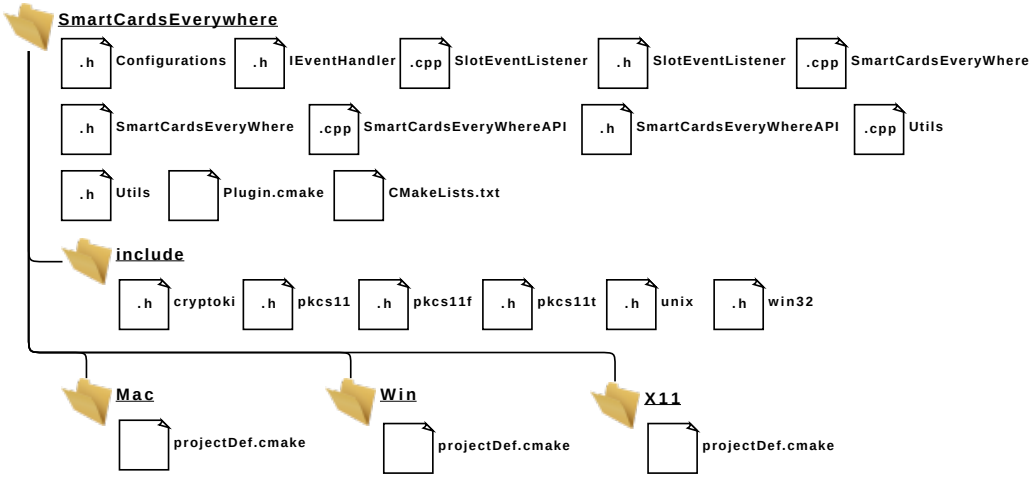


Figure 3.5: File Structure of the Plugin

Cross-platform Support

During the development phase we made a great effort to program our plugin with support for several platforms. Due to this reason we made an extensive use of the preprocessor directives `#if - #else`. These directives help us identify the platform where the plugin is being compiled. With this knowledge we had the ability to chose the most appropriate behaviour according to the target OS.

In Example 3.10 there is a precise function where we use those directives in order to know which kind of *mutexes* should be used (Microsoft Windows or POSIX). This function is used to create a *mutex*, and it is passed as a reference to the initialization function of the PKCS #11 library. The PKCS #11 library needs several *mutex*-related functions when it is being accessed by a multi-threaded application. For that purpose it needs the following four kind of functions: create and destroy a mutex, and lock and unlock a mutex.

```
//creates a mutex object
CK_RV Utilities::myCreateMutex(CK_VOID_PTR_PTR ppMutex)
{
    //mutex functions return value
    int ret = 0;
#if defined(WIN32) && !defined(UNIX)
    *ppMutex = (HANDLE *) CreateMutex(NULL,              // default security attributes
                            FALSE,             // initially not owned
                            NULL);             // unnamed mutex
    if(*ppMutex == NULL)
          ret = -1;

#else
    ret = pthread_mutex_init((pthread_mutex_t *) *ppMutex, NULL);
#endif
    return (ret == 0)? CKR_OK : CKR_GENERAL_ERROR;
}
```

Example 3.10: Cross platform support creating mutex

In the source code of our project it is possible to find several instances of `#if - #else` preprocessor directives to identify the OS, specifically to:

- choose the appropriate functions for *mutex* management;
- choose the appropriate functions for dynamic linking.
- choose the appropriate header files;

3.5 Plugin Usage

The best way to understand the behaviour and the correct use of our plugin is to present a short example of how a web application developer can take advantage of the available features.

In this section, we show how to load the plugin in a web browser whenever a user connects to a web application, and we give a little usage example of how one can use the plugin to create a DS of a file.

The first step to integrate a web application with our plugin is to reference it in the *HTML* code of the web application. This reference is accomplished using the *HTML* tag `object`, where its type must be set to `application/x+smartcardseverywhere`. Example 3.11 has an extract of a web application that is referencing our plugin. In this example we use a DOM event to get a notification once the plugin is loaded. After referencing the plugin in the *HTML*, one should create a JS function to simplify the calls to the plugin, as in Example 3.12. Now, one can perform calls to the plugin simply by using `plugin().<FUNCTION>` instead of `document.getElementById('plugin0').<FUNCTION>`.

```
1 (...)
2 <body>
3     (...)
4     <object id="plugin0" type="application/x-smartcardseverywhere" width="0" height="0">
5         <param name="onload" value="pluginLoaded"/>
6     </object>
7     (...)
8 </body>
9 (...)
```

Example 3.11: Loading the Plugin into a Web Application

```
1 function plugin0()
2 {
3     return document.getElementById('plugin0');
4 }
5 plugin = plugin0;
```

Example 3.12: Simplying the Calls to the Plugin

At this point, once the web application is loaded into a web browser, the plugin will be accessible through JS. However, it is not yet ready to perform operations or get informations from SCs, because one must explicitly instruct the plugin to initialize the PKCS #11 module. Such operation will conduct the plugin to load and prepare the PKCS #11 module to be used, as in Example 3.13. Once the plugin is not needed anymore, one may instruct it to release all resources and close all connections to SCs and the PKCS #11 module, simply by calling `plugin().finalize()`.

As we can see from Example 3.13, the call to the plugin is surrounded by a `try-catch` statement. Due to the fact that any call to the PKCS #11 module may not succeed, the plugin must warn web applications of such situations, and for that purpose it is used *exceptions*. The error that raised the exception should be inspected using the function

```
try
{
    plugin().initialize("path/to/the/pkcs11/module");
} catch(e)
{
    var error  = plugin().getLastException();
}
```

Example 3.13: Initializing the Plugin

`getLastException()` of the plugin, because it gives the exact cause of the error and it gives a better cross-browser support, since Google Chrome does not show the exceptions thrown by the plugin in the variable of the `catch` statement. In the exception that is thrown there is information regarding the function that was called, the id of the error, and an indication if it is still possible to use the plugin without initializing it. In Example 3.14 there is an instance of such exception.

```
"Function ID -> 1 ; Function Name -> finalize ; Error ID -> 400 ; Error String -> CKR_CRYPTOKI_NOT_INITIALIZED ; Remains Consistent -> 0"
```

Example 3.14: Structure of an Exception thrown by the Plugin

Complete Usage Example

Up to this point we introduced how the plugin could be successfully loaded into the web browser, and properly initialized. Now it is time to expose a usual scenario where a web application creates a SC from a file. In the following paragraphs we enumerate the steps that are needed to accomplish this goal.

1. Initialize. As we stated before the goal of this step is to properly load and initialize the PKCS #11 module, as shown in Example 3.13.

2. Get available tokens. Now that the *Cryptoki* is initialized, one should get the list of available cryptographic devices. The method `getAvailableTokens` can be used for this operation, because it returns a list of integers, where each element identifies a unique token.

3. Get available *X.509* Public Key Certificates At this point we have the list of available tokens, and we want to get additional information about each one of them. Therefore, we can use the field *subject* of a *X.509* Public Key Certificate to know the owner of each device. However, a token can have several *X.509* Public Key Certificates, like the PCC. So,

we must get the list of available items in order to get additional information about each one. The method `getAvailableX509PublicKeyCertificates` can be used in the process of getting the available *X.509* Public Key Certificates. This method has one parameter that identifies the token where the plugin should look for these items.

4. Get *X.509.* Public Key Certificate information At this step we have the list of available *X.509* Public Key Certificates for each available token in the computer. The function `getX509PublicKeyCertificateInfo` can be used in order to get the information about each certificate. This function takes two parameters, an integer that identifies the token, and another integer that identifies the certificate. The return of this function is a list of strings, where each string identifies the field of the certificate and its value, for instance: `"CKA_ID -> 0x45"`.

5. Get available private keys. Now that we now which devices are available as well as their owners, we must known which private keys are suitable to create a DS, and which mechanisms they support. But first we need to get the available private keys, since one device can have several of these. For such purpose we can use the function `getAvailablePrivateKeys`, which receives an integer indicating the token where the plugin should look for private keys.

6. Get private key information. With the list of available private keys for each device we get in step *5*, we can inspect each one for additional informations, such as the supported mechanisms. The function responsible for that operation is `getPrivateKeyInfo`, takes two parameters, and returns the list of fields of the private key. The first parameter identifies the token and the second the private key. The return is similar to the one of `getX509PublicKeyCertificateInfo`.

7. Get mechanism information. If one wishes to get additional information about a mechanism, he may do so using the function `getMechanismInfo`. This functions takes one parameter that identifies the mechanism and returns all the available data regarding it.

8. Sign the file. At this moment all the information that we need to create a DS is collected: token, private key, and mechanism. Then, we can digitally sign a file using the method `signFile`. This method takes five parameters. The first parameter identifies the token where the DS must be performed. The second identifies the private key. The third identifies the mechanism that must be used. The fourth indicates if the PIN should be asked either by the plugin or by the SC. The fifth parameter is a string containing the path to

the file that needs to be signed. The result of this method is a string containing a *base 64* encoding of the DS.

9. Finalize. Finally, the DS was successfully created and we do not need the plugin anymore, thus the plugin should release all resources and close the any established connections with SCs using the method `finalize`.

Example 3.15 summarizes all the steps we described before. For ease of representation we removed the `try-catch` blocks from the example.

```
1  //1. Initialize
2  plugin().initialize("path/to/the/pkcs11/module");
3
4  //2. Get available tokens
5  plugin().getAvailableTokens();
6
7  //3. Get availa X.509 Public Key Certificates
8  plugin().getAvailableX509PublicKeyCertificates(0);
9
10 //4. Get X.509 Public Key Certificate Information
11 plugin().getX509PublicKeyCertificateInfo(0,70);
12
13 //5. Get available private keys
14 plugin().getAvailablePrivateKeys(0);
15
16 //6. Get private key information
17 plugin().getPrivateKeyInfo(0,70);
18
19 //7. Get mechanism information
20 plugin().getMechanismInfo(0,8);
21
22 //8. Sign the file
23 plugin().signFile(0,70,8,false,"/path/to/file");
24
25 //9. Finalize
26 plugin().finalize();
```

Example 3.15: Common Steps Towards a Digital Signature Creation

Listen for Slot Events

The slot events give web applications the ability to get notifications anytime a token is either inserted or removed from a slot. In the following example we describe how a web application developer can get such notifications. We assume that the plugin is already initialized.

1. Create a function to handle a slot event Web applications will be notified about slot events through callbacks. So, one must define a JS function to handle the event. This function should contain two parameters, the first identifies the event type (insertion / removal), and the second identifies the slot where such event occurred.

2. Register an event listener in the plugin Once the handle function is defined, it must be registered in the plugin JS interface. For that purpose it must be used one of the functions: `attachEvent` or `addEventListener`. The first function is specific for the Internet Explorer web browser, and it takes two parameters, the first is the string `onslotevent` and the second is the handle function. The second function is `addEventListener` and it must be used in the remaining web browsers, it takes three parameters: the first is the string `slotevent`, the second the handle function, and the third is false.

3. Instruct the plugin to start listening slot events Now that a handle function is defined and registered in the plugin, we can instruct the plugin to start listening for slot events using the function `startListeningSlotEvents`. This functions takes no arguments and its purpose is to tell the plugin to catch any SC insertion or removal from a slot.

4. Instruct the plugin to stop Listening slot events Once someone is done listening for slot events he can use the function `stopListeningSlotEvents`, which takes no arguments and tells the plugin to stop all the process of catching and reporting these events.

5. Remove the event listener from the plugin Finally, if the web application developer wishes to unregister the handle function from the plugin, he can do so using one of the functions: `detachEvent` or `removeEventListener`. The first function can only be used in the Microsoft Internet Explorer, and it has two arguments, the first is the string `slotevent` and the second the is the handle function. The second function must be used in the remaining web browsers and has the same arguments as `detachEvent`.

In order to exemplify the steps we described in the previous paragraphs, in Example 3.16 it can be found the JS source code needed to listen for slot events.

3.6 Plugin Experimentation

According to our initial goals, we propose a new mechanism to enable SC-related features in web applications. For this mechanism we defined that it should be formed by two elements: a JS API, and a web browser plugin to handle the JS request. Furthermore, we defined that the plugin should be able to run in Google Chrome and Internet Explorer.

```
1  //1. Create a JavaScript function to handle a slot event
2  function onSlotEvent(eventID, slotID)
3  {
4      alert("A slot event occurred.\nSlot ID: "+slotID+"\nEvent ID: "+eventID);
5  }
6
7  //2. Register an event listener in the plugin
8      if($.browser.msie)//Internet Explorer?
9          plugin().attachEvent("onslotevent", onSlotEvent);
10     else//Other web browsers
11         plugin().addEventListener("slotevent", onSlotEvent, false);
12
13 //3. Start listening for slot events
14     plugin().startListeningSlotEvents();
15
16 //4. Stop listening slot events
17     plugin().stopListeningSlotEvents();
18
19 //5. Remove event listener from the plugin
20     if($.browser.msie)//Internet Explorer?
21         plugin().detachEvent("slotevent",onSlotEvent);
22     else//Other web browsers
23         plugin().removeEventListener("slotevent",onSlotEvent);
```

Example 3.16: Enabling Slot Events in a Web Application

Supported Web Browsers and Operating Systems

As we described throughout this chapter our main efforts were to create a browser-independent mechanism, that could be easily ported between web browsers. In that sense, we tested our plugin in the following three web browsers that according to a statistic of World Wide Web Consortium (W3C)[24] are the most used:

- Google Chrome;
- Microsoft Internet Explorer;
- Mozilla Firefox.

For each one of these web browsers we were able to install and test our plugin under the following OSs:

- LUbuntu 12.04;
- Microsoft Windows XP Professional SP3;
- Mac OS X Snow Leopard.

In Section A.3 there are several images exposing our plugin being in the available plugins for each one of these web browsers.

[24] http://www.w3schools.com/browsers/browsers_stats.asp

In our work we did not implement any features that are specific for a given platform our web browser. So, we expect that our plugin may be also compatible with other web browsers supported by the *Firebreath* framework, specifically the ones that support the NPAPI interface.

For each platform and web browser where we were succeed installing our plugin we could test all the plugin features. In these tests the following features showed the same results among all web browsers and platforms we tested:

- inspect SCs to get the stored information, like X.509 Public Key Certificates;
- perform cryptographic operations, namely DSs and digests;
- notify web applications of changes in slots through DOM-like events.

Supported Smart Cards

The case study of our project was the PCC. As we already describe in other sections, the vendor of this SC — the Portuguese Government — supplies an implementation of the PKCS #11 standard, and this was the implementation we mainly used in our tests.

Using our plugin we were able to test the features of the PCC, specifically creation of DSs and digests, and check its contents, specifically the available X.509 Public Key Certificates.

At the present moment we expect that our plugin is able to work with other implementations of the PKCS #11 standard, since all the features we implemented are not specific to PCC. All the available features and their implementation were designed using the official reference manual of PKCS #11. Therefore, we expect that our plugin may work with different SCs, besides the PCC. For that purpose, the implementation of *OpenSC* of the PKCS #11 standard can enable our plugin to work with the following SCs:

- Estonian electronic identification card;
- German electronic identification card.

3.6.1 Output Examples

In Example 3.17 we can see the result of a DS. In this small example it was used the function `signData` to perform the cryptographic operation. The first step in this operation was to encode the text that was going to be signed in the *Base 64* representation. For that purpose it can be used the function `toBase64` available in the JS interface of the plugin. On the right side of this example there is the created DS. The configuration for this operation was the following:

- Token: 0
- Private key: 0x45
- Mechanism: 8 (`CKM_RIPEMD160_RSA_PKCS`)
- Prompt pin: false
- Data to sign: *Base 64* representation of `This is just some random text`

JavaScript Instructions

```
var textToSign = plugin()
                    .toBase64("This is just
                              some random
                              text");

plugin().signData(0,
                  0x45,
                  8,
                  false,
                  textToSign);
```

Digital Signature

```
"UpXP+P+is8+TCTDCtFiqC+lsBDglNkcjFu0dAI8e
NOgfw/JRoYFSKi5qKyY2ZFsqrD9sAQBGV38/iMFJb
xPUo9D0TU6f4JAInem9Ild9n8gZAXUVeufkn8vedf
BvaSxvv9st6HlhHnSZ8w2L8tJyDfoeRkheWv3UKq3
j1dEnUn8="
```

Example 3.17: Output Example of a Digital Signature

The Example 3.18 shows the result of an inspection for additional informations regarding a X.509 Public Key Certificate. On the left side there is the request that was performed to the JS interface of the plugin. The function `getX509PublicKeyCertificateInfo` was used in this process to get the details of the X.509 Public Key Certificate identified as `0x45` in the token `0`. On the right side there is an extract of the result. The first field of this extract indicates the label of the certificate, and the second one shows the subject encoded in the Distinguished Encoding Rules (DER) format.

In order to understand the DER encoded data of the subject of the X.509 Public Key Certificate, we show in Example 3.19 the result of decoding this data. As we can see, this certificate is issued to *Leonel João Fernandes Braga*.

JavaScript Instructions

```
plugin()
    .getX509PublicKeyCertificateInfo(0,
                                     0x45);
```

Extract of the X.509 Public Key Certificate

```
[(...),
 "CKA_LABEL ->
    CITIZEN AUTHENTICATION CERTIFICATE",
 (...)
 "CKA_SUBJECT ->
    MIHcMQswCQYDVQQGEwJQVDEcMBoGA1UECgwTQ2F
    ydMOjbyBkZSBDaWRhZMOjbzEjMCEGA1UECwwaQX
    V0ZW50aWNhw6fDo28gZG8gQ2lkYWTDo28xHDAaB
    gNVBAsME0NpZGFkw6NvIFBvcnR1Z3XDqnMxGDAW
    BgNVBAQMD0ZFUk5BTkRFUyBCUkFHQTEVMBMGA1U
    EKgwMTEVPTkVMIEpPw4NPMRQwEgYDVQQFEwtCST
    EzMzIxNjkyNjElMCMGA1UEAwwcTEVPTkVMIEpPw
    4NPIEZFUk5BTkRFUyBCUkFHQQAA",
 (...)
]
```

Example 3.18: Ouput Example of a X.509 Public Key Certificate

```
SEQUENCE(8 elem)
    SET(1 elem)
        SEQUENCE(2 elem)
            OBJECT IDENTIFIER 2.5.4.6
            PrintableString PT
    SET(1 elem)
        SEQUENCE(2 elem)
            OBJECT IDENTIFIER 2.5.4.10
            UTF8String Cartão de Cidadão
    SET(1 elem)
        SEQUENCE(2 elem)
           OBJECT IDENTIFIER2.5.4.11
            UTF8String Autenticação do Cidadão
    SET(1 elem)
        SEQUENCE(2 elem)
            OBJECT IDENTIFIER2.5.4.11
            UTF8String Cidadão Português
    SET(1 elem)
        SEQUENCE(2 elem)
            OBJECT IDENTIFIER2.5.4.4
            UTF8String FERNANDES BRAGA
    SET(1 elem)
        SEQUENCE(2 elem)
            OBJECT IDENTIFIER2.5.4.42
            UTF8String LEONEL JOÃO
    SET(1 elem)
        SEQUENCE(2 elem)
            OBJECT IDENTIFIER2.5.4.5
            PrintableString BI133216926
    SET(1 elem)
        SEQUENCE(2 elem)
            OBJECT IDENTIFIER2.5.4.3
            UTF8String LEONEL JOÃO FERNANDES BRAGA
```

Example 3.19: Subject of a X.509 Public Key Certificate

3.7 Summary

The process of building a web browser plugin to successfully expose SCs features in web applications was described in this chapter. We started by presenting the decisions we had to make in order to design our solution. Then, we presented the design the JS interface that we though would best serve the needs of web application developers. Among the several available methods in the JS interface, web application developers can find many ways to inspect the contents of SCs and create DS. In this step we decided to hide some details about the PKCS #11 library, for instance, getting information about a private key object requires only the use of a single function (`getPrivateKeyInfo`), instead of the usual three in PKCS #11: `C_FinObjectsInit` + `C_FindObjects` + `C_GeAttributeValue`. The creation of DS is another good example where we hide the PKCS #11 complexity, in first place all the connections and sessions to the SC are controlled by the plugin, and the feeding mechanism of the signing mechanism as well.

As we showed in Sections 3.3 and 3.4, *Firebreath* made the creation of the JS interface easier, it gave us the possibility to expose methods, attributes, and events with few lines of code, and with strong typing from JS values to C++ types, which ensures a correct matching of data between these two languages, and a more secure execution.

Finally, we were able to compile, deploy and test the plugin among several web browsers and OSs, with no restrictions, and we were also able to execute all the features in those systems, as well.

Chapter 4

Security Analysis

Up to this point we discussed the available techniques that could be used to access Smart Cards (SCs) and develop web browser plugins. Then, we used this knowledge to decide from the available options what should be our mechanisms to implement the web browser plugin and access SCs. The design details of the JavaScript (JS) interface of the plugin, and its implementation were also reviewed as well.

Now we are going to perform an exploratory analysis of the security of the plugin. In order to perform this task we will resort to tools and techniques that can help us identify and address possible problems.

The first task we performed in this analysis was to check for problems in the plugin source code using tools for static analysis. In Section 4.1 we present the result of this work, and discuss the additional analysis that could be performed but we did not have the opportunity to put in place.

In Section 4.2 we use Attack Trees [Schneier, 1999] to model the several attacks that can be perpetrated by malicious agents to achieve a certain goal. A discussion of the more relevant attacks and possible counter-measures is presented as well.

In order to understand the quality of the source code we developed, we analysed its maintainability in Section 4.3. In this task we used a model proposed in [Heitlager et al., 2007]. This analysis could give us an insight of the quality of maintainability characteristics, like: analysability and changeability. From these characteristics we can understand if the source code is easy to analyse. Auditing of source code can be easier to perform when it is easy to analyse. Therefore, it would be interesting to assess the maintainability of our project.

4.1 Source Code Analysis

At the present moment there are several available tools to analyse software in order to identify problems like memory leaks. These tools can be divided in two main categories: static and dynamic. Usually, in a static analysis the program is not executed. The tool performing this analysis checks for problems in the source code of the program. In a dynamic analysis the program is executed, and usually the problems are detect in behaviour of the program. This analysis requires that the inputs to the program cover a great majority of the source code, so every aspect of the program is checked.

Static Analysis

In our project we decided to run tools for static analysis in order to mitigate any problem in the plugin. So, the first step was to find suitable tools for this purpose. In our search we discarded all the tools with commercial license, thus we end up choosing:

- **CPPCheck**[1] **-** is a tool for finding bugs in software, and it is very useful to find problems like: memory leaks, bounds checking, exception safety, check input/output operations, use of deprecated or unsafe functions.
- **Flawfinder**[2] **-** is a tool that analyses the source code in order to find security weaknesses.

From the tools we found to perform static analysis of software, these were the only ones that: are free; can be applied to C++ programs, and can be used to find problems in the source code.

The output of these tools helped us find and address one problem regarding an eventual buffer overflow, and an incorrect use of the function `sprintf`.

Dynamic Analysis

Unfortunately, due to time restrictions we did not have the time to perform other analysis besides static. Performing a dynamic analysis in a web browser plugin is more complex than in a ordinary program. Since the web browser plugin runs in a process of the web browser, performing a dynamic analysis would require to isolate (and intercept) the plugin from the web browser.

[1] http://sourceforge.net/apps/mediawiki/cppcheck/index.php
[2] http://www.dwheeler.com/flawfinder/

A dynamic analysis would give us a real insight of the plugin behaviour. The results of such analysis can be used to find the presence of memory leaks, software bugs, and even if the software meets its requirements. In order to perform such analysis we would create a third-party program that would be formed by the source code responsible for handling the JS requests. In this program we could run several types of dynamic analysis like:

- *Unit testing*
- *Fuzz testing*

In a fuzz testing technique a program is subjected to unexpected inputs. Usually, these inputs are automatically generated and are not by any means valid. So, the goal for this test is to monitor the program according to such inputs and check for problems like memory leaks and software crashes. For instance, software crashes due to dangling pointer references can be used by attackers to overwrite arbitrary memory locations [Younan et al., 2004]. Fuzz testing techniques are typically used to find security flaws in software.

In this scenario, tools for dynamic analysis like *Valgrind* could be used as well. We may use this software tool to check for memory leaks in the ordinary execution of the functions responsible for treating the JS requests.

4.2 Attack Trees

The attack trees provide a way to model threats against computer systems in a formal and methodical way [Schneier, 1999]. The first element that needs to be identified in this model is the goal of the attack. This element is going to be the root element of the tree. Then, we should identify the different ways of achieving that goal. Each different way will be leaf node of the tree. The Figure 4.1 illustrates an attack tree that identifies some ways that can be used to open a safe.

In attack trees it is also possible to estimate costs if a given goal is achieved, or calculate how probable it is to achieve such goal. In order to estimate such values we must attach the each leaf node of the tree the cost (or probability) of the attack. As we can see in Figure 4.1 the estimated cheapest cost of attack is $10K.

In the mentioned example it was used cost, but other values could be attached to the nodes as well. For instance, one may attach to each leaf node of the tree: probabilities, risk categories (low, moderate, high), possibility (possible, impossible), skills.

An attack tree can be reused, that is, it is possible to use an attack tree as a node of another one. This is very handy when you have a large system and there are several people analysing its security, and when there is an attack that can be used in several attack trees of a system.

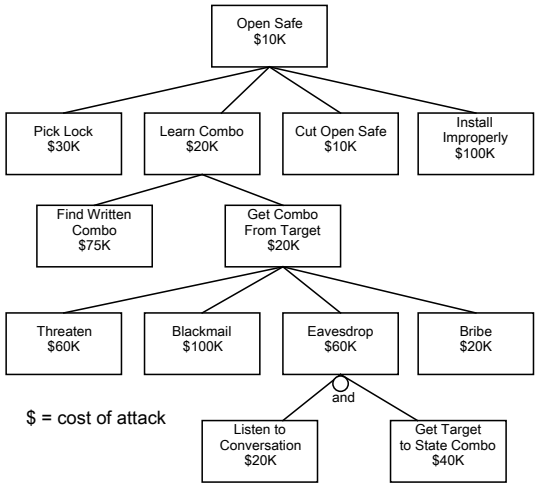


Figure 4.1: Attack Tree - Open Safe. Adapted from [Schneier, 1999]

4.2.1 Modelling Possible Attacks to the Plugin

Throughout this part we will present and discuss the attack trees we created during the security analysis. These attack trees will expose the several threats that can be perpetrated to each one of the following goals we identified:

- create digital signature,
- collect user data,
- compromise the plugin.

Create Digital Signature

The cryptographic properties of Digital Signatures (DSs) offer a precise way to provide unforgeable proof of identity. As we already discussed, DSs can be used to prove identity in order to authenticate a user in a service, and to prove the origin of files. From these examples we can conclude that the access to the mechanisms that create DSs must be very well protected. An open vulnerability can easily led malicious agents to perform a valid DS using someone else's identity. Therefore, if this mechanism is not well protected, stealing user's identity can be achieved using our plugin.

Figure 4.2 presents the attack tree we created for the attack goal: create DS. As we can see, we identified three main threats to this goal: mislead the user, compromise the plugin, learn pin.

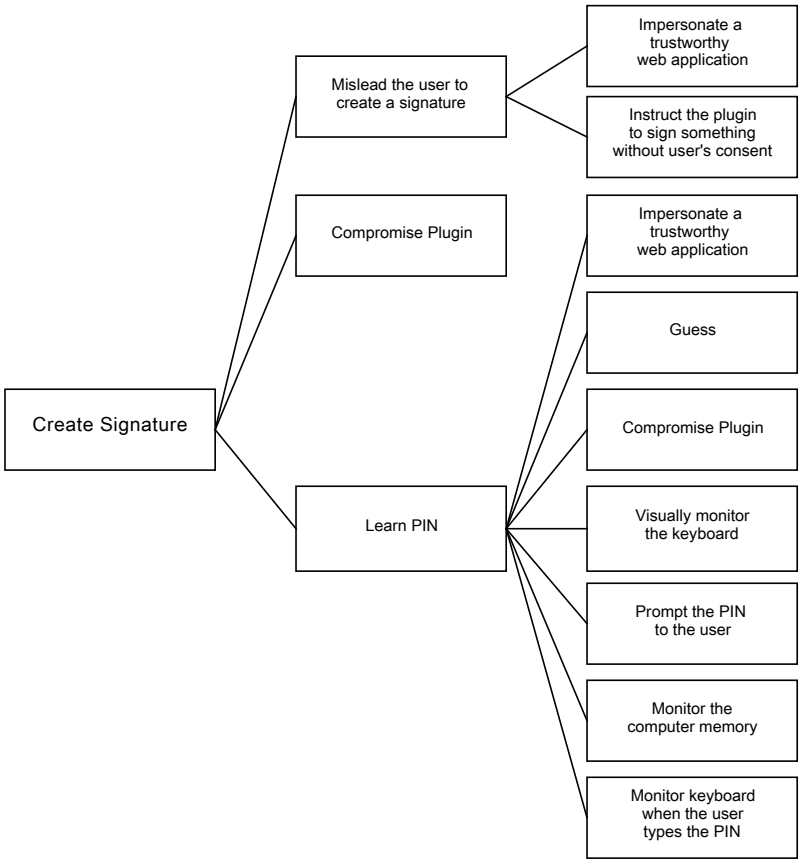


Figure 4.2: Attack Tree - Create Digital Signature

For the first threat we imagined a scenario where an attacker deceives the user into creating a DS. In this scenario he can trick the user to access a web application that may seem trustworthy. A better description of this scenario is described in another attack tree. The attacker can also order the plugin to sign something that the user is not aware of. For instance, sign binary data is very risky. In order to minimize such risk, our plugin always prompts if the user wishes to continue with the operation.

An attacker may also attack the plugin implementation in order to get privileged access to the SC. For instance, replacing the Public-Key Cryptography Standards #11 (PKCS #11) library may enable an attacker to have full control over the SC. A more detailed discussion over this attack is detailed in another attack tree.

If an attacker gains access to the Personal Identification Number (PIN) that protects the private key, he could easily create a DS. However, in our plugin that would not be enough. In our implementation of the creation of a DS we always ask the user if he wishes to continue with the operation, and the plugin always prompts the PIN. Web applications do not have direct access to PKCS #11 library, thus they do not have a way to enter the PIN in order to create a DS.

Collect User Data

Malicious agents may not only be interested in creating DSs: gathering personal and private data may be desired as well. The Figure 4.3 presents the threats we identified to this attack. The major risks come from compromising the plugin and lure users to access harmful web applications. Details of these attacks are presented in another attack trees.

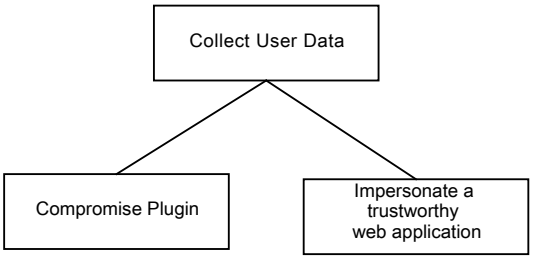


Figure 4.3: Attack Tree - Collect User Data

Compromise the Plugin

A successful attack to the plugin implementation can compromise not only the plugin itself, but of all the system. The threats we identified to this goal are shown in Figure 4.4.

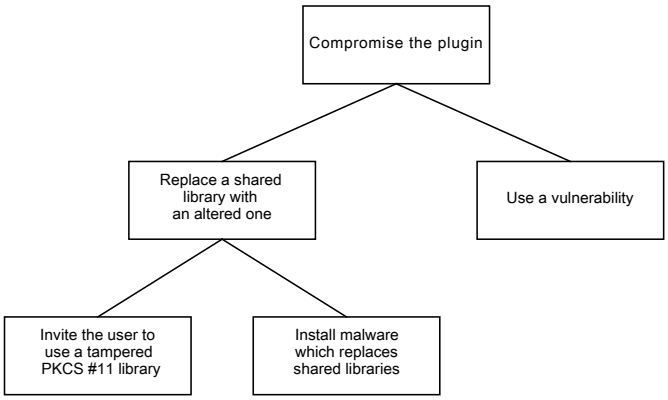


Figure 4.4: Attack Tree - Compromise the Plugin

At the present moment one of the major risks of the plugin is the linking with the PKCS #11 module, and any other shared library. A malicious agent which replaces one of those libraries can easily have access to all the system. Since we are linking the plugin to PKCS #11 modules at runtime, this can be easily used to compromise the plugin. One may lure a client to install and use a tampered PKCS #11 module. In order to prevent such attacks we could link the plugin statically. To complement this protection we could use Code Signing techniques

provided by web browsers and operating systems (OSs) to prevent malicious agents from tampering the plugin. Dynamic linking could be used as well, but additional measures must be taken. For instance, we could sign the plugin, and load only PKCS #11 modules that are code signed by known and trustworthy entities.

A vulnerability in the plugin can also be used to gain access to the system. Unfortunately we did not have the opportunity to perform a more detailed analysis to the plugin source code and execution.

Impersonate a Trustworthy Web Application

The security of a user can be broke without attacking directly the plugin. Attackers may lure users to access unreliable web applications in order to steal their identities, access their data, and even create DSs.

The Figure 4.5 reflects some of attacks that can be made in order to use the plugin as if the web application were reliable.

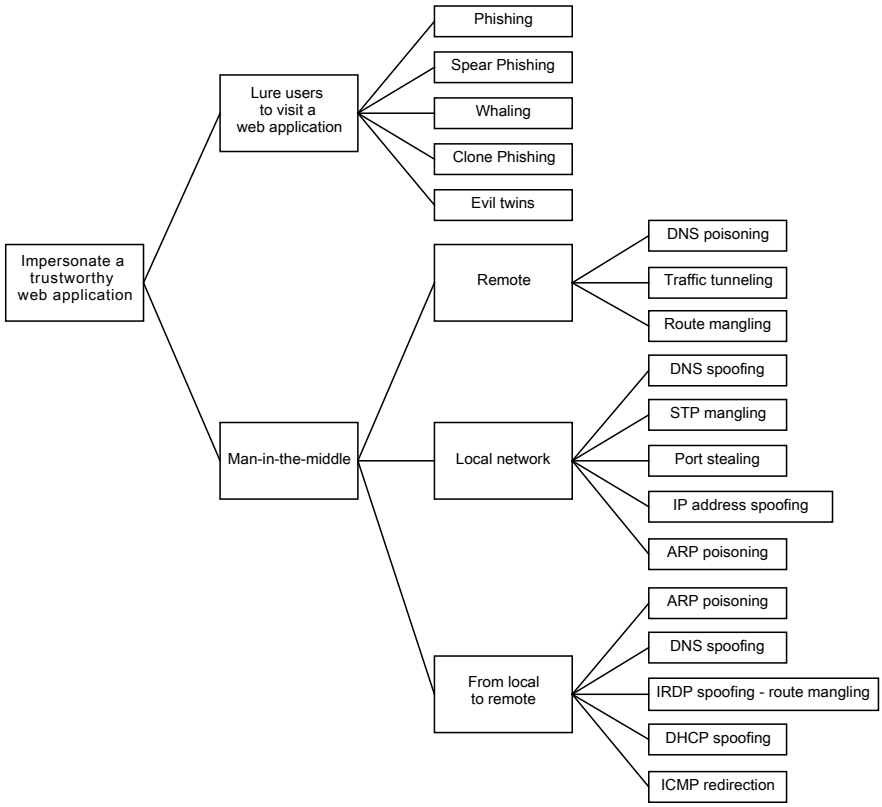


Figure 4.5: Attack Tree - Impersonate a Trustworthy Web Application

The simplest attacks do not require a high level of expertise from the attacker. To perpetrate such attacks the agent may use social engineering skills to lure users to visit a web application that may seem trustworthy. These attacks are all related to phishing schemes, and they may be easy to implement when the connection is not protected. In order to prevent these scenarios the plugin could enforce secure *HTTP* connections with known and trustworthy sources. For instance, if a web application wishes to use the plugin, the connection must be through *HTTPS* and it must have a valid and known X.509 Public Key Certificate. Otherwise, the plugin would simply not handle any request from the JS interface.

Another attack that can be used to achieve the same goal is Man-in-the-middle. If the plugin enforces a secure connection between client and server in order to handle JS requests, some sort of these attacks may be prevented. Even if the attacker manages to put himself between web application and client, he would not have ways to prove to the client and the plugin that he holds a valid X.509 Public Key Certificate. However, an attack to the connection using a method like *BEAST*[3] can led an attacker to eavesdrop the communication.

Final remarks

The security analysis using attack trees to model goals and threats is far from complete. This is just an initial step towards a global understanding of the system. The mechanism that we developed is destined to be used by web-based systems, thus we must ensure security in the four fronts: web client, data transport, web server, and operating system [Skoularidou and Spinellis, 2003]. In that sense, we should not only perform an more detailed audit to the plugin source code and to its execution, but to the surrounding system. From such analysis we can then understand how the system affects the plugin security, and vice-versa, in order to create a higher protection for users and web applications

4.3 Maintainability Analysis

The result of measuring the maintainability of software can be used to understand the complexity of a program. As mentioned in [Goldberg et al., 1996, Seacord, 2008], avoid and identify bugs it is easier in simple and maintainable programs. Due to that reason we decided to measure the maintainability of the plugin.

At the present moment there are several models that can be used to measure maintainability. Specifically, the *ISO 9126* defines several characteristics that influence maintainability. However, it does not provide a consensual way to estimate the maintainability using source code

[3] http://www.schneier.com/blog/archives/2011/09/man-in-the-midd_4.html

properties [Heitlager et al., 2007]. In this project we decided the maintainability of our plugin using the model by *SIG* [Heitlager et al., 2007]. We chose this model because in this work its authors created a match between the characteristics of the maintainability defined in the *ISO 9126* and the source code properties. In this model, source code properties are traced back to the sub-characteristics of the maintainability defined in the *ISO 9126*: analysability, changeability, stability, and testability. From these sub-characteristics we can identify the aspects of the source code that can influence security:

- The *analysability* indicates how easy or difficult it is to understand the source code. Therefore, it is easier to find deficiencies and identify the parts that need to be modified when the source code is not complex.

- The *changeability* indicates how easy or difficult it is to create modifications in the source code. It is desired that changes in source code are easy to perform once security vulnerabilities are found.

- The *stability* indicates how easy or difficult it is to keep the system in a consistent state during modification.

- The *testability* indicates how easy it is to test the program. Software testing is very important to find problems in programs, thus find security vulnerabilities.

The definitions of the characteristics we presented above were adapted from [Heitlager et al., 2007].

The first step when analysing the maintainability using the *SIG* model is to measure several source code properties. These properties will be measure with respect to specific source code metrics. The properties and how they are measured are described next:

- **Volume -** is intended to express the size of the project. Since a project can be developed using many programming languages, in this model it must be used: Man Years (MY) via backfiring function points.

- **Complexity per unit -** is intended to express how complex are source code units. A unit can be a function, a method. This property is measure using Lines of Code (LOC).

- **Duplication -** is intended to express the presence of repeated lines of code in the project. In order to measure this property, it must be counted the number of repeated blocks of at least six lines of code, discarding comments and blank lines.

- **Unit size -** is intended to express the average size of the source code units. The source code metric that is used to measure this property is LOC.

- **Unit testing -** is intended to express how tested the system is. The source code metric that is used to measure this property is the coverage of the unit tests.

After measuring each one of these source code properties, we can map them onto the characteristics of the maintainability, as show in Table 4.1. For instance, the changeability is affected by the complexity per unit and the duplication. Each one the source code properties will be ranked in a scale that varies from 1 to 5, being 1 the worst rank.

ISO 9126 Maintainability	Source Code Properties: Volume	Complexity per Unit	Duplication	Unit Size	Unit Testing
analysability	x		x	x	x
changeability		x	x		
stability					x
testability		x			x

Table 4.1: Mapping between the characteristics of the maintainability and source code properties. Adapted from [Heitlager et al., 2007].

Volume

As we already described, the Volume is the first source code property to measured. In the first place we counted per language the LOC of the project. The results of this step are show in Table 4.2.

Plugin	**LOC**
C++	4653
Sum	4653

Table 4.2: Lines of Code per Language in the Plugin Source Code

The second step is to convert the LOC of each language to MY, and then sum the MY of each language. In Table 4.3 there is the conversion factor for C++, and the sum of MY of the plugin.

Finally, it is time to rank this source code property in our plugin. In Table 4.4 we show how the rank must be evaluated, and we present the decision for our project. As we can see, our project is assessed as ++, the highest level. This result was expected because this is a small project.

Language	Conversion Factors to MY	Plugin
C++	11458	0.406
Sum		0.406

Table 4.3: Conversion Factors to Man Years, and Man Years per Language of the Plugin Source Code

Rank	Man Years	Plugin MY	Plugin Rank
++	0 - 8	0.406	++
+	8 - 30		
0	30 - 80		
-	80 - 160		
−−	> 160		

Table 4.4: Evaluation of the Volume Metric of the Plugin Source Code

Complexity per Unit

The complexity per unit is the second source code property to be measured. The first operation in this step is to count the cyclomatic complexity of each function of the source code. Then, we must sum the LOC of each function that fit in each category of the table Table 4.5. For instance, in the first category must be the sum of the LOC of all functions that have a cyclomatic complexity lesser than 10. In this process we decided to exclude some functions that were partially generated automatically. This is why the sum of the LOC is different from the LOC presented in the Volume.

Cyclomatic Complexity	Risk Evaluation	Plugin Relative LOC	Plugin Relative LOC (%)
1 - 10	simple, without much risk	1319	80.62
11 - 20	more complex, moderate risk	167	10.21
21 - 50	complex, high risk	150	9.17
> 50	untestable, very high risk	0	0

Table 4.5: Categories of Risks in Complexity per Unit

After grouping the LOC through each one of the categories it is time to rank the source code property: Complexity per Unit. The Table 4.6 exposes the criteria that must be used in order to assess this source code property. For instance, a project to be ranked as ++ in this property must have a maximum of 25% of LOC in the moderate category, and 0% in the high and very high categories. Our plugin is ranked as 0 in this source code property.

Coincidentally the functions with a high LOC also have a high cyclomatic complexity. In that sense, these functions contribute to a higher percentage of LOC, thus increasing the percentage of LOC with a high risk.

Rank	Maximum Relative LOC (%)			Plugin Rank
	Moderate (%)	High (%)	Very High (%)	
++	25	0	0	0
+	30	5	0	
0	40	10	0	
-	50	15	5	
−−	-	-	-	

Table 4.6: Ranking the Complexity per Unit

Duplication

In order to rank the duplication in our plugin we had to measure the number of repeated blocks with more than 5 LOC (discarding comments and blank lines). Using this value it is calculated the percentage of repeated LOC. The Table 4.7 we present the results for our plugin. As we can see we did not find any repeated block in our source code.

Duplicated Lines	0
% Duplicated Lines	0.00%

Table 4.7: Repeated Lines of Code in the Plugin Source Code

After counting the repeated blocks the rank for the duplication can be calculated. The table Table 4.8 exposes the criteria that must be used to assess this property. According to this model our plugin is ranked as ++ in this property. During the development phase we tried to reuse all the functions and source code in order to reduce repeated lines of code.

Rank	Duplication	Plugin Duplication	Plugin Rank
++	0 - 3%	0.0	++
+	3 - 5%		
0	5 - 10%		
-	10 - 20%		
−−	20 - 100%		

Table 4.8: Ranking the Duplication

Unit Size

The unit size property is used to understand the average size of the functions of a program. The first step is to identify how many LOC fit in each one of the categories shown in Table 4.9.

The second step is to rank the property. For that purpose we use the criteria exposed in Table 4.10. This criteria tell us that a program to be ranked in this property as + must have

Unit Size	Risk Evaluation	Plugin Relative LOC	Plugin Relative LOC (%)
0 - 20	Low	446	27.26
21 - 50	Moderate	480	29.34
51 - 100	High	609	37.22
> 100	Very High	101	06.17

Table 4.9: Categories of Risks in Unit Size

Rank	Maximum Relative LOC			Plugin Rank
	Moderate (%)	High (%)	Very High (%)	
++	25	0	0	–
+	30	5	0	
0	40	10	0	
-	50	15	5	
−−	-	-	-	

Table 4.10: Ranking the Unit Size

a maximum LOC of 30% in the moderate risk, 5% in the high risk, and 0% in the very high risk. From this criteria we can conclude that our pluginis ranked as −−.

Although in the development phase we made a great effort to create small functions, the verbosity of the PKCS #11 did not always helped. Handling with the PKCS #11 standard requires the use of several functions to achieve a simple goal like: retrieve the list of available private keys. We did our bests to reuse source code, but there were several times when such approach could not be applied. Splitting functions into smaller ones could be an alternative, but these new functions would not be used in other places. Another reason why we end up having some functions with a high and very high risk is due to the object inspection. Since we are returning all the data available to a given object, the functions in charge of that operation have several lines of code dedicated to define attributes that must be returned. As we already discussed, the introduction of templates in object inspection is a attractive scenario that should be implemented in the future, and it help in reducing the LOC per unit as well.

Coverage

The last source code property to be ranked is the coverage of the unit tests. The first step in this operation is to calculate the percentage of lines of code covered by the unit tests. In Table 4.11 there is the result for our project. As we mentioned previously, due to time restrictions we did the opportunity to exercise the plugin through unit tests.

The Table 4.12 shows the criteria that must be used in order to assess this source code property. As we can see our plugin has the lowest score: −−. This is obviously a deficiency

Lines Covered	0
% Lines Covered	0.00%

Table 4.11: Lines of Code Covered

in our development, and a field where a additional efforts must be made in order to assure that the plugin behaves has expected and that runtime flaws are addressed.

Rank	**Coverage**	**Plugin Coverage**	**Plugin Rank**
++	95 - 100%	0.0	−−
+	80 - 95%		
0	60 - 80%		
-	20 - 60%		
−−	0 - 20%		

Table 4.12: Ranking the Coverage

Overall

Now that all source code properties were properly ranked, we can trace back these properties to the characteristics of maintainability. This relation between source code properties and the characteristics of the maintainability are expressed in Table 4.13. This one of the main advantages of the model by *SIG*: we can trace source code metrics to properties of maintainability in a pratical and precise way.

		Source Code Properties					
		Volume	Complexity per Unit	Duplication	Unit Size	Unit Testing	
		++	0	++	−−	−−	
ISO 9126 Maintainability	analysability	x		x	x	x	0
	changeability		x	x			+
	stability					x	−−
	testability		x		x	x	-

Table 4.13: Overall Results of the Maintainability Analysis

In this final step in order to measure the maintainability we can assess the rank of each characteristic. For that purpose we can estimate the average of the ranks obtained for each source code property that affect each characteristic. For instance, the changeability is affected

by the complexity per unit and duplication. The results for each characteristic is express in the rightmost column of Table 4.13.

From these results we can enumerate the following conclusions:

- The analysability of the plugin can be improved. In order to accomplish that goal we should reduce the average size of the functions and implement unit tests. This last option is one of the must crucial, because it affects many other characteristics of the maintainability, and it would eventually improve security.

- Eventual changes in the source code of the plugin would be easy to implement. One of the aspects that influence this characteristic is the absence of duplicated source code. As we already mentioned when discussing the assessment of complexity per unit, reducing the average unit size would also reduce the complexity per unit. Therefore, the changeability would also be improved.

- With the conclusion of this analysis it became more obvious that unit tests need to be implemented. Their implementation increase the stability of the plugin.

- Finally, implementing unit tests is not enough. Reducing the complexity per unit and the unit size would allow us to achieve a higher source code coverage, thus increasing testability as well.

4.4 Summary

As stated in [Schneier, 1999]: "Security is not a product — it's a process". In that sense, the work we performed in the security analysis of the plugin is just a first step towards a full comprehension of risks and vulnerabilities.

The static analysis of the plugin source code helped us identify the misuse of few functions and address one case of an eventual buffer overflow. Nevertheless, additional analysis must be performed in order to complement this work. Dynamic analysis are some instances of techniques that we did not have the opportunity to implement, but that would offer a major contribute towards identifying vulnerabilities in the plugin. In those techniques we can include unit testing, and fuzz testing.

The attack trees played a major roll in identifying possible goals of attacks and threats, in a methodical and formal way. This analysis allowed us identify the greater risks in the plugin security, and discuss possible measures that can be taken in a future work to address such

problems. As we mentioned, the major risks came from the use of non secure *HTTP* connections, and the linking with the PKCS #11 library.

Finally, the maintainability analysis made us realise exactly which source code properties must be enhanced. For instance, we must decrease the average size of functions and implement unit tests. The unit size affects analysability, and a high analysability eases eventual audit tasks to source code. Unit tests affects stability and testability, and a high level of testing can help identify eventual problems in the plugin execution.

Chapter 5

Conclusion

The absence of a common mechanism which enables Smart Card (SC) features in web applications has led to the creation of distinct solutions to address such limitations. These solutions are often similar regarding the type of operations they perform, and sometimes they lack portability, forcing users to move away from the web browser and operating system (OS) they are accustomed.

The intent of this project was to solve these limitations through the development of an uniform accessible mechanism to SCs across web browsers. For this purpose we decided to create a web browser plugin to: (1) connect the web browser to SCs, and (2) to expose the SC related features to web applications through a JavaScript (JS) Application Programming Interface (API).

According to the initial goals, the developed plugin can indeed successfully perform all the operations defined at the beginning of the project, and thus several SC functionalities are available to web applications. At this moment, it is possible to retrieve a variety of information from a SC, such as: *Public Key Certificates*, supported mechanisms, and available *Private Keys*. Regarding cryptographic operations, it is available in the JS of the plugin two kinds of functions which will be performed by SCs. The first kind of functions enables the creation of a Digital Signature (DS) from either a file, or a blob of bytes. The second type of cryptographic functions can be used to create hashes — or digests — from also either files or bytes.

Regarding plugin development, we should emphasize the adoption of the *Firebreath* framework as our build platform. Adopting it allowed us to focus solely on building the plugin, therefore saving time explicitly supporting the platforms *NPAPI* and *Activex Control*, and creating mechanisms to build the plugin for several OSs.

The plugin security was reviewed, as well. In that phase of the project we tried to identify vulnerabilities in the source code and in the plugin usage. For that purpose, we resorted to

static analysis tools in order to find problems like *buffer overflows*, and we used *Attack Trees* to create a model of which kind of attacks can be perpetrated. The static analysis revealed very few warnings with a low risk, which were immediately resolved. After some runs that analysis did not show evidences of further errors. The *Attack Trees* helped us realize that the main category of attacks come from luring users into download malicious software or to use the plugin in an untrustworthy web application. Some of these attacks can be addressed if measures like the ones mentioned in the next section are taken.

Future Work

The mechanism that we developed may be a first prototype of an effort towards an unification of methods to access SC from web applications, but there are several aspects where it must be improved, either regarding of security, or feature enhancement.

One of the most important steps in future developments of this work could be an extensive software testing, using unit tests, dynamic analysis tools, and testing frameworks. These techniques give us the ability to verify if the software complies with its requirements, and allow us to address any identified vulnerability, thus improving the security of the plugin.

Although we are not aware, at this moment, of any method to directly check a web browser plugin behaviour using unit tests, there are other methods to achieve the same result. For instance, we can create a dynamic library from the source code in charge of processing the JS requests, in order to exercise their internal behaviour using unit tests. Then, the JS API can be tested using Jasmine[1], a framework for testing JS. Besides unit tests, it would be very important to use other kind of techniques like fuzz testing, which has an essential role checking software behaviour for random inputs of data.

The use of dynamic analysis tools can give us further insights into the plugin execution. For instance, *Valgrind*[2] is a powerful tool which can detect memory management and threading bugs. Once again, fixing an error —in this case what would be a runtime error— can enhance the plugin security.

SeleniumHQ[3] is a tool for automated tests in web browsers which can be used in conjunction with a testing web application for the plugin to check the plugin behaviour.

As discussed in the security analysis of the plugin, the major security risks are related with the distribution of the plugin. A malicious agent can easily lure users to install a fake plugin

[1] http://pivotal.github.com/jasmine
[2] http://valgrind.org/
[3] http://seleniumhq.org/

in order to get access to their system. Therefore the use of Code Signing techniques is certainly one of the major features to implement in future releases. These techniques can be used to sign the plugin itself and the Public-Key Cryptography Standards #11 (PKCS #11) modules. As stated in Section 2.4, all major web browsers and OSs have support for this feature. *Firebreath* has also available automated mechanisms to create a code-signed *ActiveX Control*, which we did not explore.

Open vulnerabilities in software can be exploited by attackers to get access to users' computers. One may use a software flaw to execute restricted code that otherwise would not be allowed. Migrating the plugin into a Sandbox could restrict the application permissions, so malicious code would not have access beyond the allowed area. At this moment, only Google Chrome has a framework for plugin development using *Sandboxing*. A complementary project would be to implement *Sandboxing* in the remaining web browsers.

Man-in-the-middle attacks can be easily implemented to collect transmitted data between users and servers, when the connection is not safe. So, enforcing secure *HTTP* connections, even when the user dismisses the web browser warnings for a possibly untrusted server identity, can address some of those attacks. This solution can solve some of phishing schemes attacks, as well.

In terms of features, it would be interesting if the plugin could provide an additional module that could be used to interact with the *eID Lib API* from the Portuguese Citizenship Card (PCC). Such interaction would give web applications the ability to access more information stored in the SC, such as: address, age, parentage, gender. In case this information is exposed to web applications, additional security measures would be needed, in order to protect the users' identity. We can include in those measures the creation of a trust network, where only web applications with a trustworthy identity could access such information.

As we describe in Subsection 2.2.1, the PKCS #11 standard defines a template-based model to access objects and object attributes stored inside SCs. In Section 3.1 we explained why we chose to hide such model from web application developers, and the reason why we decided to expose all available attributes of each object. However, adopting a template-based model for the object attributes would be more efficient and concise.

References

C. Adams and S. Lloyd. *Understanding PKI: concepts, standards, and deployment considerations*. Technology series. Addison-Wesley, 2003. ISBN 9780672323911. URL http://books.google.com/books?id=ERSfUmmthMYC. **Cited** on pages 1 and 2.

Agência para a Modernização Administrativa. Manual técnico do middleware cartão de cidadão. Technical report, Agência para a Modernização Administrativa, Julho 2007. URL http://www.cartaodecidadao.pt/images/stories/manual%20t%E9cnico%20do%20middleware%20do%20cc_v1%200.pdf. **Cited** on pages 12, 13 and 41.

Agência para a Modernização Administrativa. Autenticação com o cartão de cidadão. Technical report, Agência para a Modernização Administrativa, Dezembro 2008. URL http://www.cartaodecidadao.pt/images/stories/Manual%20Autenticacao%20com%20Cartao%20de%20Cidadao_%20v1.7.pdf. **Cited** on page 2.

Jean-Daniel Aussel. Smart cards and digital security. In Vladimir Gorodetsky, Igor Kotenko, and Victor A. Skormin, editors, *Computer Network Security*, volume 1 of *Communications in Computer and Information Science*, pages 42–56. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-73986-9. URL http://dx.doi.org/10.1007/978-3-540-73986-9_4. **Cited** on page 1.

A T Chan. WWW + smart card: towards a mobile health care management system. *International Journal of Medical Informatics*, 57(2-3):127–137, 2000. URL http://www.ncbi.nlm.nih.gov/pubmed/10961569. **Cited** on page 8.

Alvin T. S. Chan. Integrating smart card access to web-based medical information systems. In *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, pages 246–250, New York, NY, USA, 2003. ACM. ISBN 1-58113-624-2. doi: http://doi.acm.org/10.1145/952532.952583. URL http://doi.acm.org/10.1145/952532.952583. **Cited** on page 8.

Alvin T. S. Chan, Jiannong Cao, Henry Chan, and Gilbert Young. A web-enabled framework for smart card applications in health services. *Commun. ACM*, 44:76–82, September 2001. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/383694.383710. URL http://doi.acm.org/10.1145/383694.383710. **Cited** on page 8.

Dipankar Dasgupta, Sudip Saha, and Aregahegn Negatu. Techniques for Validation and Controlled Execution of Processes, Codes and Data - A Survey. In *Security and Cryptography*, pages 77–85, 2010. URL http://ieeexplore.ieee.org/xpl/login.jsp?arnumber=5741635. **Cited** on pages 20, 23 and 27.

Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *USENIX Security Symposium*, 1996. URL http://static.usenix.org/publications/library/proceedings/sec96/full_papers/goldberg/goldberg.pdf. **Cited** on pages 25, 26 and 76.

Ilja Heitlager, Tobias Kuipers, and Joost Visser. A Practical Model for Measuring Maintainability. In *International Conference on the Quality of Information and Communications Technology*, pages 30–39, 2007. doi: 10.1109/QUATIC.2007.8. URL http://ieeexplore.ieee.org/xpl/login.jsp?arnumber=4335232. **Cited** on pages 6, 69, 77 and 78.

H K Lu and A M Ali. Making smart cards truly portable. *Security Privacy IEEE*, 8(2):28–34, 2010. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5416672. **Cited** on pages 1 and 9.

H Karen Lu, Asad Ali, Kapil Sachdeva, and Ksheerabdhi Krishna. A pragmatic online authentication framework using smart cards. *Online*, (c):84–91, 2011. **Cited** on page 9.

H.K. Lu, A.M. Ali, S. Durand, and L. Castillo. A new secure communication framework for smart cards. In *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, pages 1 –5, jan. 2009. doi: 10.1109/CCNC.2009.4784726. **Cited** on page 9.

John R. Michener and Tolga Acar. Managing System and Active-Content Integrity. *IEEE Computer*, 33:108–110, 2000. doi: 10.1109/2.869389. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=869389. **Cited** on page 22.

Vassilis Prevelakis and Diomidis Spinellis. Sandboxing Applications. In *USENIX Technical Conference*, pages 119–126, 2001. URL http://static.usenix.org/publications/library/proceedings/usenix01/freenix01/full_papers/prevelakis/prevelakis.pdf. **Cited** on pages 24 and 25.

W. Rankl and W. Effing. *Smart Card Handbook*. John Wiley & Sons, 2004. ISBN 9780470856680. URL http://books.google.pt/books?id=Oi85gPhUFx4C. **Cited** on page 1.

RSA Laboratories. RSA Security Inc. Public-Key Cryptography Standards (PKCS). Technical report, RSA Laboratories, 2004. URL ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20.pdf. **Cited** on pages 11, 13, 14, 41 and 103.

Aviel D. Rubin and Daniel E. Geer Jr. Mobile Code Security. *IEEE Internet Computing*, 2:30–34, 1998. doi: 10.1109/4236.735984. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=735984. **Cited** on page 20.

Kapil Sachdeva, H Karen Lu, and Ksheerabdhi Krishna. A Browser-Based Approach to Smart Card Connectivity. In *IEEE Workshop on Web 2.0 Security and Privacy*, Oakland, California, 2009. URL http://w2spconf.com/2009/papers/s4p4.pdf. **Cited** on pages 7, 9, 10, 11 and 12.

Damien Sauveron. Multiapplication smart card: Towards an open smart card? *Inf. Secur. Tech. Rep.*, 14:70–78, May 2009. ISSN 1363-4127. doi: 10.1016/j.istr.2009.06.007. URL http://dl.acm.org/citation.cfm?id=1595066.1595093. **Cited** on pages 2, 8 and 28.

Jay Schiavo. Code signing for end-user peace of mind. *Network Security*, 2010:11–13, 2010. doi: 10.1016/S1353-4858(10)70093-3. URL http://www.sciencedirect.com/science/article/pii/S1353485810700933. **Cited** on page 20.

B. Schneier. Attack trees - modeling security threats. 1999. URL http://www.schneier.com/paper-attacktrees-ddj-ft.html. **Cited** on pages 6, 69, 71, 72 and 83.

R.C. Seacord. *The Cert C Secure Coding Standard*. Sei Series in Software Engineering. Addison-Wesley, 2008. ISBN 9780321563217. URL http://books.google.pt/books?id=6ipFVfxKeN0C. **Cited** on pages 6 and 76.

George Selimis, Apostolos Fournaris, George Kostopoulos, and Odysseas Koufopavlou. Software and Hardware Issues in Smart Card Technology. *IEEE Communications Surveys & Tutorials*, 11: 143–152, 2009. doi: 10.1109/SURV.2009.090310. URL http://ieeexplore.ieee.org/xpl/login.jsp?arnumber=5208738. **Cited** on page 1.

Victoria Skoularidou and Diomidis Spinellis. Security architectures for network clients. *Information Management & Computer Security*, 11:84–91, 2003. doi: 10.1108/09685220310468664. URL http://www.dmst.aueb.gr/dds/pubs/jrnl/2003-IMCS-clisec/html/cli-sec.pdf. **Cited** on pages 22 and 76.

Guenther Starnberger, Lorenz Froihofer, and Karl M. Goeschka. A generic proxy for secure smart card-enabled web applications. In *Proceedings of the 10th international conference on Web engineering*, ICWE'10, pages 370–384, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13910-8, 978-3-642-13910-9. URL http://dl.acm.org/citation.cfm?id=1884110.1884141. **Cited** on page 8.

Robert Wahbe, Steven Lucco, and Thomas E. Anderson. Efficient Software-Based Fault Isolation. *Operating Systems Review*, 27:203–216, 1993. doi: 10.1145/168619.168635. URL http://crypto.stanford.edu/cs155old/cs155-spring07/sfi.pdf. **Cited** on page 25.

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy*, volume 53, pages 79–93, 2009. doi: 10.1109/SP.2009.25. URL http://johmathe.nonutc.fr/ressources/nacl_paper.pdf. **Cited** on page 28.

Yves Younan, Wouter Joosen, and Frank Piessens. Code injection in c and c++ : A survey of vulnerabilities and countermeasures. Technical report, DEPARTEMENT COMPUTERWETENSCHAPPEN, KATHOLIEKE UNIVERSITEIT LEUVEN, 2004. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.2429. **Cited** on page 71.

Appendix A

Documentation

Throughout this chapter it is possible to find following documentation that we created during this project:

Section A.1 has a detailed description of the JS interface of the plugin. In this section we describe each member of the interface. It is possible to find some output examples of the plugin as well.

Section A.2 shows several tables which aggregate information of each one of the available objects defined in the PKCS #11 standard.

Section A.3 exposes images showing our plugin installed in several web browsers.

A.1 JavaScript API

In this section we present all the methods, attributes, and events available on the JS API of the plugin.

In order to explain with a level of detail the behaviour of each one of the several members of the JS interface of the plugin, we decided to included the expected types.

Method(s)			
init	**Signature**	`void init(string pkcs11location)`	
	Category	Environment	
	Description	This method must be used in order to load and initialize the PKCS#11 module	
	Returns	—	
	Throws Exception	Yes	
	Pre Condition	—	
	Post Condition	Success: the PKCS#11 module is loaded and ready to use	
	Parameter(s)	`string pkcs11location`	This parameter must have a valid path to the location of the PKCS#11 module

finalize	**Signature**	`void finalize()`
	Category	Environment
	Description	This method must be used once the plugin is not needed anymore
	Returns	—
	Throws Exception	Yes
	Pre Condition	The plugin must be initialized
	Post Condition	Success: all the resources are released
start Listening Slot Events	**Signature**	`void startListeningSlotEvents()`
	Category	Environment
	Description	Instructs the plugin to start listening for slot events
	Returns	—
	Throws Exception	Yes
	Pre Condition	The plugin must be initialized
	Post Condition	Success: the plugin is waiting for events in all slots
stop Listening Slot Events	**Signature**	`void stopListeningSlotEvents()`
	Category	Environment
	Description	Instructs the plugin to stop listening for slot events
	Returns	—
	Throws Exception	Yes
	Pre Condition	The plugin must be initialized and listening for slot events
	Post Condition	Success: the plugin stops listening for slot events
get Library Information	**Signature**	`string[] getLibraryInformation()`
	Category	Inspection
	Description	Gets information about the PKCS#11 module
	Returns	—
	Throws Exception	Yes
	Pre Condition	The plugin must be initialized
	Post Condition	Success: information about the PKCS#11 module is retrieved to the web application
get Available Tokens	**Signature**	`int[] getAvailableTokens()`
	Category	Inspection
	Description	Gets the available tokens
	Returns	Returns an array of integers, where each element identifies exactly a token
	Throws Exception	Yes
	Pre Condition	The plugin must be initialized
	Post Condition	Success: a list of available tokens is returned to the web application
get Token Information	**Signature**	`string[] getTokenInformation(int tokenID)`
	Category	Inspection
	Description	Gets information about a particular token
	Returns	Returns an array of strings, where each element has a pair attribute + value, example: `label-->abc`
	Throws Exception	Yes
	Pre Condition	The plugin must be initialized and the tokenID must be a valid identifier
	Post Condition	Success: a list containing the information about the token is returned
	Parameter(s)	`int tokenID` | Identifies the token where the operation must be performed

get Available Mechanisms	**Signature**	`int[] getAvailableMechanisms()`	
	Category	Inspection	
	Description	Gets the available cryptographic mechanisms supported by the Smart Card	
	Returns	Returns an array of integers, where each element identifies exactly one mechanism	
	Throws Exception	Yes	
	Pre Condition	The plugin must be initialized	
	Post Condition	Success: a list of available mechanisms is returned	
get Mechanism Info	**Signature**	`string[] getMechanismInfo(int tokenID, int mechID)`	
	Category	Inspection	
	Description	Gets information about a given mechanism supported by a given token	
	Returns	Returns an array of strings, where each element is a pair attribute + value of the mechanism	
	Throws Exception	Yes	
	Pre Condition	The plugin must be initialized, and the token and mechanism identifiers must be valid	
	Post Condition	Success: information about the mechanism is returned	
	Parameter(s)	`int tokenID`	Identifies the token where the operation must be performed
		`int mechID`	Identifies the mechanism
get Available Private Keys	**Signature**	`int[] getAvailablePrivateKeys(int tokenID)`	
	Category	Inspection	
	Description	Gets information about the available private keys on given cryptographic device	
	Returns	Returns an array of integers, where each element identifies exactly a private key	
	Throws Exception	Yes	
	Pre Condition	The plugin must be initialized, and the token identifier must be valid	
	Post Condition	Success: information regarding the available private keys is returned	
	Parameter(s)	`int tokenID`	Identifies the token where the operation must be performed
get Private Key Info	**Signature**	`string[] getPrivateKeyInfo(int tokenID, int privKeyID`	
	Category	Inspection	
	Description	Gets information about a particular private key of a given token	
	Returns	Returns an array of strings, where each element is a pair attribute + value of the private key	
	Throws Exception	Yes	
	Pre Condition	The plugin must be initialized, and the identifiers of the token and private key must be valid	
	Post Condition	Success: information about the private key is returned	
	Parameter(s)	`int tokenID`	Identifies the token where the operation must be performed
		`int privKeyID`	Identifies the private key

get Available Public Keys	**Signature**	`int[] getAvailablePublicKeys(int tokenID)`	
	Category	Inspection	
	Description	Gets information about the available public keys on a given cryptographic device	
	Returns	Returns an array of integers, where each element identifies exactly a public key	
	Throws Exception	Yes	
	Pre Condition	The plugin must be initialized, and the identifiers must be valid	
	Post Condition	Success: a list containing the available public keys is returned	
	Parameter(s)	`int tokenID`	Identifies the token where the operation must be performed
get Public Key Info	**Signature**	`string[] getPublicKeyInfo(int tokenID, int pubKeyID)`	
	Category	Inspection	
	Description	Gets information about a particular public key of a given token	
	Returns	Returns an array of strings, where each element is a pair attribute + value of the public key	
	Throws Exception	Yes	
	Pre Condition	The plugin must be initialized, and the identifiers must be valid	
	Post Condition	Success: a list containing the details of the public key is returned	
	Parameter(s)	`int tokenID`	Identifies the token are the operation must be performed
		`int pubKeyID`	Identifies the public key
get Available X509 Public Key Certificates	**Signature**	`int[] getAvailableX509PublicKeyCertificates(int tokenID)`	
	Category	Inspection	
	Description	Gets list of available X.509 Public Key Certificates in a given token	
	Returns	Returns an array of integers, where each element identifies exactly one object	
	Throws Exception	Yes	
	Pre Condition	The plugin must be initialized and the token identifier must be valid	
	Post Condition	Success: a list containing the available elements is returned	
	Parameter(s)	`int tokenID`	Identifies the token where the operation must be performed
get X509 Public Key Certificate Info	**Signature**	`string[] getX509PublicKeyCertificateInfo(int tokenID, int certID)`	
	Category	Inspection	
	Description	Gets information about a particular X.509 Public Key Certificate on a given cryptographic device	
	Returns	Returns an array of strings, where each element is a pair attribute + value of the certificate	
	Throws Exception	Yes	
	Pre Condition	The plugin must be initialized and the identifiers must be valid	
	Post Condition	Success: a list containing the certificate details is returned	
	Parameter(s)	`int tokenID`	Identifies the token where the operation must be performed
		`int certID`	Identifies the certificate

get Available WTLS Public Key Certificates	**Signature**	`int[] getAvailableWTLSPublicKeyCertificates(int tokenID)`
	Category	Inspection
	Description	Gets a list of available WTLS Public Key Certificates
	Returns	Returns an array of integers, where each element identifies exactly one certificate
	Throws Exception	Yes
	Pre Condition	The plugin must be initialized and the identifier must be valid
	Post Condition	Success: a list of available certificates is returned
	Parameter(s)	`int tokenID` Identifies the token where the operation must be performed
get WTLS Public Key Certificate Info	**Signature**	`string[] getWTLSPublicKeyCertificateInfo(int tokenID, int certID)`
	Category	Inspection
	Description	Gets information about a particular WTLS Key Certificate on a given cryptographic device
	Returns	Returns an array of strings, where each element is a pair attribute + value of the certificate
	Throws Exception	Yes
	Pre Condition	The plugin must be initialized and the identifiers must be valid
	Post Condition	Success: a list containing the certificate details is returned
	Parameter(s)	`int tokenID` Identifies the token where the operation must be performed
		`int certID` Identifies the certificate
get Available X509 Attribute Certificates	**Signature**	`int[] getAvailableX509AttributeCertificates(int tokenID)`
	Category	Inspection
	Description	Gets a list of available X.509 attribute Certificates
	Returns	Returns an array of integers, where each element identifies exactly one certificate
	Throws Exception	Yes
	Pre Condition	The plugin must be initialized and the identifier must be valid
	Post Condition	Success: a list of available certificates is returned
	Parameter(s)	`int tokenID` Identifies the token where the operation must be performed
get X509 Attribute Certificate Info	**Signature**	`string[] getX509AttributeCertificateInfo(int tokenID, int certID)`
	Category	Inspection
	Description	Gets information about a particular X.509 Attribute Certificate on a given cryptographic device
	Returns	Returns an array of strings, where each element is a pair attribute + value of the certificate
	Throws Exception	Yes
	Pre Condition	The plugin must be initialized and the identifiers must be valid
	Post Condition	Success: a list containing the certificate details is returned
	Parameter(s)	`int tokenID` Identifies the token where the operation must be performed
		`int certID` Identifies the certificate

get Available Data Objects	**Signature**	`int[] getAvailableDataObjects(int tokenID)`	
	Category	Inspection	
	Description	Gets a list of available data objects on a given cryptographic device	
	Returns	Returns an array of integers where each element identifies exactly one data object	
	Throws Exception	Yes	
	Pre Condition	The plugin must be initialized and the identifier must be valid	
	Post Condition	Success: a list containing the available data objects is returned	
	Parameter(s)	`int tokenID`	Identifies the token where the operation must be performed
get Data Object Info	**Signature**	`string[] getDataObjectInfo(int tokenID, int objID)`	
	Category	Inspection	
	Description	Gets information about a particular data object on a given cryptographic device	
	Returns	Returns an array of strings, where each element is a pair attribute + value of the data object	
	Throws Exception	Yes	
	Pre Condition	The plugin must be initialized and the identifiers must be valid	
	Post Condition	Success: a list containing the details of the data object is returned	
	Parameter(s)	`int tokenID`	Identifies the token where the operation must be performed
		`int objID`	Identifies the data object
sign Data	**Signature**	`string signData(int tokenID, int privKeyID, int mechID, bool prompPin, string data`	
	Category	Cryptographic	
	Description	Creates a digital signature of binary data	
	Returns	Returns a Base 64 encoded string containing the digital signature	
	Throws Exception	Yes	
	Pre Condition	The plugin must be initialized and the identifiers must be valid.	
	Post Condition	Success: a digital signature is returned	
	Parameter(s)	`int tokenID`	Identifies the token where the operation must be performed
		`int privKeyID`	Identifies the private key that must used to create the digital signature
		`int mechID`	Identifies the mechanism that must be used to generate the digital signature
		`bool prompPin`	True if the plugin must prompt the PIN to the user. False if the cryptographic device has mechanisms for PIN prompting
		`string data`	A Base 64 encoded string containing the binary data that is supposed to sign

sign File	**Signature**	`string signFile(int tokenID, int privKeyID, int mechID, bool prompPin, string path2file`	
	Category	Cryptographic	
	Description	Creates a digital signature of a file	
	Returns	Returns a Base 64 encoded string containing the digital signature	
	Throws Exception	Yes	
	Pre Condition	The plugin must be initialized and the identifiers must be valid. The path must be valid and point to a file	
	Post Condition	Success: a digital signature is returned	
	Parameter(s)	`int tokenID`	Identifies the token where the operation must be performed
		`int privKeyID`	Identifies the private key that must used to create the digital signature
		`int mechID`	Identifies the mechanism that must be used to generate the digital signature
		`bool prompPin`	True if the plugin must prompt the PIN to the user. False if the cryptographic device has mechanisms for PIN prompting
		`string path2file`	The path to the file that is supposed to be signed
digest Data	**Signature**	`string digestData(int tokenID, int mechID, string data`	
	Category	Cryptographic	
	Description	Creates a digest of binary data	
	Returns	Returns a Base 64 encoded string containing the digest	
	Throws Exception	Yes	
	Pre Condition	The plugin must be initialized and the identifiers must be valid	
	Post Condition	Success: a digest from the binary data is returned	
	Parameter(s)	`int tokenID`	Identifies the token where the operation must be performed
		`int mechID`	Identifies the mechanism that must be used to generate the digital signature
		`string data`	A Base 64 encoded string containing the binary data
digest File	**Signature**	`string digestFile(int tokenID, int mechID, string path2file`	
	Category	Cryptographic	
	Description	Creates a digest of a file	
	Returns	Returns a Base 64 encoded string containing the digest	
	Throws Exception	Yes	
	Pre Condition	The plugin must be initialized and the identifiers must be valid. The path must be valid and point to a file	
	Post Condition	Success: a digest from the file is returned	
	Parameter(s)	`int tokenID`	Identifies the token where the operation must be performed
		`int mechID`	Identifies the mechanism that must be used to generate the digital signature
		`string path2file`	The path to the file that is supposed to be signed

to Base 64	**Signature**	`string toBase64(string data)`
	Category	Utilities
	Description	Encodes a string into the Base 64 format
	Returns	Returns a Base 64 encoded string of the input data
	Throws Exception	Yes
	Pre Condition	—
	Post Condition	Success: the data is encoded to the base 64 and returned
	Parameter(s)	`string data` | the data to encode
from Base 64	**Signature**	`string fromBase64(string data)`
	Category	Utilities
	Description	Decodes a string in the Base 64 format
	Returns	Returns a string containing the decoded data
	Throws Exception	Yes
	Pre Condition	The data must be properly encoded in the Base 64 format
	Post Condition	Success: the decoded data is returned
	Parameter(s)	`string data` | the base 64 encoded data

Attribute(s)

FIRST_ STRING_ SEPARATOR	Type	`string`
	Category	String Separators
	Rean-only	true
SECOND_ STRING_ SEPARATOR	Type	`string`
	Category	String Separators
	Rean-only	true
THIRD_ STRING_ SEPARATOR	Type	`string`
	Category	String Separators
	Rean-only	true
MULTIPLE_ VALUES_ SEPARATOR	Type	`string`
	Category	String Separators
	Rean-only	true
CKF_ RNG	Type	`unsigned integer`
	Category	Token flags
	Rean-only	true
CKF_ WRITE_ PROTECTED	Type	`unsigned integer`
	Category	Token flags
	Rean-only	true
CKF_ LOGIN_ REQUIRED	Type	`unsigned integer`
	Category	Token flags
	Rean-only	true
CKF_ USER_ PIN_ INITIALIZED	Type	`unsigned integer`
	Category	Token flags
	Rean-only	true
CKF_ RESTORE_ KEY_ NOT_ NEEDED	Type	`unsigned integer`
	Category	Token flags
	Rean-only	true

CKF_ CLOCK_ ON_ TOKEN	Type	unsigned integer
	Category	Token flags
	Rean-only	true
CKF_ PROTECTED_ AUTHENTICATION_ PATH	Type	unsigned integer
	Category	Token flags
	Rean-only	true
CKF_ DUAL_ CRYPTO_ OPERATIONS	Type	unsigned integer
	Category	Token flags
	Rean-only	true
CKF_ TOKEN_ INITIALIZED	Type	unsigned integer
	Category	Token flags
	Rean-only	true
CKF_ SECONDARY_ AUTHENTICATION	Type	unsigned integer
	Category	Token flags
	Rean-only	true
CKF_ USER_ PIN_ COUNT_ LOW	Type	unsigned integer
	Category	Token flags
	Rean-only	true
CKF_ USER_ PIN_ FINAL_ TRY	Type	unsigned integer
	Category	Token flags
	Rean-only	true
CKF_ USER_ PIN_ LOCKED	Type	unsigned integer
	Category	Token flags
	Rean-only	true
CKF_ USER_ PIN_ TO_ BE_ CHANGED	Type	unsigned integer
	Category	Token flags
	Rean-only	true
CKF_ SO_ PIN_ COUNT_ LOW	Type	unsigned integer
	Category	Token flags
	Rean-only	true
CKF_ SO_ PIN_ FINAL_ TRY	Type	unsigned integer
	Category	Token flags
	Rean-only	true
CKF_ SO_ PIN_ LOCKED	Type	unsigned integer
	Category	Token flags
	Rean-only	true
CKF_ SO_ PIN_ TO_ BE_ CHANGED	Type	unsigned integer
	Category	Token flags
	Rean-only	true
CKF_ HW	Type	unsigned integer
	Category	Mechanism flags
	Rean-only	true
CKF_ ENCRYPT	Type	unsigned integer
	Category	Mechanism flags
	Rean-only	true

CKF_ DECRYPT	Type	unsigned integer
	Category	Mechanism flags
	Rean-only	true
CKF_ DIGEST	Type	unsigned integer
	Category	Mechanism flags
	Rean-only	true
CKF_ SIGN	Type	unsigned integer
	Category	Mechanism flags
	Rean-only	true
CKF_ SIGN_ RECOVER	Type	unsigned integer
	Category	Mechanism flags
	Rean-only	true
CKF_ VERIFY	Type	unsigned integer
	Category	Mechanism flags
	Rean-only	true
CKF_ VERIFY_ RECOVER	Type	unsigned integer
	Category	Mechanism flags
	Rean-only	true
CKF_ GENERATE	Type	unsigned integer
	Category	Mechanism flags
	Rean-only	true
CKF_ GENERATE_ KEY_ PAIR	Type	unsigned integer
	Category	Mechanism flags
	Rean-only	true
CKF_ WRAP	Type	unsigned integer
	Category	Mechanism flags
	Rean-only	true
CKF_ UNWRAP	Type	unsigned integer
	Category	Mechanism flags
	Rean-only	true
CKF_ DERIVE	Type	unsigned integer
	Category	Mechanism flags
	Rean-only	true
CKF_ EXTENSION	Type	unsigned integer
	Category	Mechanism flags
	Rean-only	true
TOKEN_ INSERTED	Type	unsigned integer
	Category	Slot Events
	Rean-only	true
TOKEN_ REMOVED	Type	unsigned integer
	Category	Slot Events
	Rean-only	true
Event(s)		

slot event	Description	This event notifies web applications any time a token is either insert/removed into/from a slot	
	Internet Explorer	onslotevent	
	Other web browsers	slotevent	
	Parameter(s)	int	Identifies the event type. 1 if a token was inserted, 2 otherwise.
		int	Identifies the slot where the event happened.

Table A.1: Documentation of the JS Interface of the Plugin

A.2 PKCS #11 Objects Reference

In the following tables we aggregated all the available information defined for each one of the objects in the PKCS #11 standard. The information shown in these tables is just a compilation of the data available in the reference manual of PKCS #11 [RSA Laboratories, 2004]. The intellectual property and credits of this information belong to the *RSA* company.

Hardware Feature Objects			
Class definition	**CK_OBJECT_CLASS**	CKO_HW_FEATURE	
Common Attributes	**Attributes**	**Data Type**	**Meaning**
	CKA_HW_FEATURE_TYPE	CK_HD_FEATURE	Hardware Feature (type)
Attributes			
CK_HD_FEATURE	**Attribute**	**Data Type**	**Meaning**
CKH_CLOCK	CKA_VALUE	CK_CHAR[16]	Current time as a character string of length 16. Format: YYYYMMDDhhmmssxx
CKH_MONOTONIC_COUNTER	CKA_RESET_ON_INIT	CK_BBOOL	The value of the counter will reset to a previously returned value if the token is initialized using C_InitializeToken.
	CKA_HAS_RESET	CK_BBOOL	The value of the counter has been reset at least once at some point in time.
	CKA_VALUE	Byte array	The current version of the monotonic counter. The value is returned in big endian order.
CKH_USER_INTERFACE	CKA_PIXEL_X	CK_ULONG	Screen resolution (in pixels) in X-axis
	CKA_PIXEL_Y	CK_ULONG	Screen resolution (in pixels) in Y-axis
	CKA_RESOLUTION	CK_ULONG	DPI, pixels per inch
	CKA_CHAR_ROWS	CK_ULONG	For character-oriented displays
	CKA_CHAR_COLUMNS	CK_ULONG	For character-oriented displays If display is of proportional-font type, this is the width of the display in "em"-s (letter "M"), see CC/PP Struct.
	CKA_COLOR	CK_BBOOL	Color support
	CKA_BITS_PER_PIXEL	CK_ULONG	The number of bits of color or grayscale information per pixel.
	CKA_CHAR_SETS	RFC 2279 string	String indicating supported character sets, as defined by IANA MIBenum sets (www.iana.org). Supported character sets are separated with ";".
	CKA_ENCODING_METHODS	RFC 2279 string	String indicating supported content transfer encoding methods, as defined by IANA (www.iana.org). Supported methods are separated with ";"
	CKA_MIME_TYPES	RFC 2279 string	String indicating supported (presentable) MIME-types, as defined by IANA (www.iana.org). Supported types are separated with ";"
Notes			
CKH_CLOCK	The CKA_VALUE attribute may be set using the C_SetAttributeValue function if permitted by the device. The session used to set the time must be logged in. The device may require the SO to be the user logged in to modify the time value. C_SetAttributeValue will return the error CKR_USER_NOT_LOGGED_IN to indicate that a different user type is required to set the value.		
CKH_MONOTONIC_COUNTER	The CKA_VALUE attribute may not be set by the client.		
CKH_USER_INTERFACE	The selection of attributes, and associated data types, has been done in an attempt to stay as aligned with RFC 2534 and CC/PP Struct as possible. The special value CK_UNAVAILABLE_INFORMATION may be used for CK_ULONG-based attributes when information is not available or applicable.		
	None of the attribute values may be set by an application.		
	The value of the CKA_ENCODING_METHODS attribute may be used when the application needs to send MIME objects with encoded content to the token.		

Table A.2: Hardware Feature Objects in PKCS # 11

Mechanism objects		
Class definition		
CK_OBJECT_CLASS	CKO_MECHANISM	
Attributes		
	Value	
Attribute	**Data Type**	**Meaning**
CKA_MECHANISM_TYPE	CK_MECHANISM_TYPE	The type of mechanism object
Notes		
	The CKA_MECHANISM_TYPE attribute may not be set..	

Table A.3: Mechanism Objects in PKCS # 11

Storage Objects		
Class definition		
CK_OBJECT_CLASS	CKO_DATA	
Attributes		
	Value	
Attribute	**Data Type**	**Meaning**
CKA_TOKEN	CK_BBOOL	CK_TRUE if object is a token object; CK_FALSE if object is a session object. Default is CK_FALSE.
CKA_PRIVATE	CK_BBOOL	CK_TRUE if object is a private object; CK_FALSE if object is a public object. Default value is token- specific, and may depend on the values of other attributes of the object.
CKA_MODIFIABLE	CK_BBOOL	CK_TRUE if object can be modified Default is CK_TRUE.
CKA_LABEL	RFC2279 string	Description of the object (default empty).
Notes		
Only the CKA_LABEL attribute can be modified after the object is created. (The CKA_TOKEN, CKA_PRIVATE, and CKA_MODIFIABLE attributes can be changed in the process of copying an object, however.)		
The CKA_TOKEN attribute identifies whether the object is a token object or a session object.		
When the CKA_PRIVATE attribute is CK_TRUE, a user may not access the object until the user has been authenticated to the token.		
The value of the CKA_MODIFIABLE attribute determines whether or not an object is read-only. It may or may not be the case that an unmodifiable object can be deleted.		
The CKA_LABEL attribute is intended to assist users in browsing.		

Table A.4: Storage Objects in PKCS # 11

Data Objects		
Class definition		
CK_OBJECT_CLASS	CKO_DATA	
Attributes		
	Value	
Attribute	**Data Type**	**Meaning**
CKA_TOKEN	CK_BBOOL	CK_TRUE if object is a token object; CK_FALSE if object is a session object. Default is CK_FALSE.
CKA_PRIVATE	CK_BBOOL	CK_TRUE if object is a private object; CK_FALSE if object is a public object. Default value is token- specific, and may depend on the values of other attributes of the object.
CKA_MODIFIABLE	CK_BBOOL	CK_TRUE if object can be modified Default is CK_TRUE.
CKA_LABEL	RFC2279 string	Description of the object (default empty).
CKA_APPLICATION	RFC2279 string	Description of the application that manages the object (default empty)
CKA_OBJECT_ID	Byte Array	DER-encoding of the object identifier indicating the data object type (default empty)
CKA_VALUE	Byte array	Value of the object (default empty)
	Notes	
	The CKA_APPLICATION attribute provides a means for applications to indicate ownership of the data objects they manage. Cryptoki does not provide a means of ensuring that only a particular application has access to a data object, however.	
	The CKA_OBJECT_ID attribute provides an application independent and expandable way to indicate the type of the data object value. Cryptoki does not provide a means of insuring that the data object identifier matches the data value.	

Sample template	CK_OBJECT_CLASS class = CKO_DATA;
	CK_UTF8CHAR label[] = "A data object";
	CK_UTF8CHAR application[] = "An application";
	CK_BYTE data[] = "Sample data";
	CK_BBOOL true = CK_TRUE;
	CK_ATTRIBUTE template[] = {
	{CKA_CLASS, &class, sizeof(class)},
	{CKA_TOKEN, &true, sizeof(true)},
	{CKA_LABEL, label, sizeof(label)-1},
	{CKA_APPLICATION, application, sizeof(application)-1},
	{CKA_VALUE, data, sizeof(data)}
	};	

Table A.5: Data Objects in PKCS # 11

Domain Parameter Objects		
Class definition		
CK_OBJECT_CLASS	CKO_DOMAIN_PARAMETERS	
Attributes		
	Value	
Attribute	**Data Type**	**Meaning**
CKA_TOKEN	CK_BBOOL	CK_TRUE if object is a token object; CK_FALSE if object is a session object. Default is CK_FALSE.
CKA_PRIVATE	CK_BBOOL	CK_TRUE if object is a private object; CK_FALSE if object is a public object. Default value is token- specific, and may depend on the values of other attributes of the object.
CKA_MODIFIABLE	CK_BBOOL	CK_TRUE if object can be modified Default is CK_TRUE.
CKA_LABEL	RFC2279 string	Description of the object (default empty).
CKA_KEY_TYPE	CK_KEY_TYPE	Type of key the domain parameters can be used to generate.
CKA_LOCAL	CK_BBOOL	CK_TRUE only if domain parameters were either - generated locally (i.e., on the token) with a C_GenerateKey - created with a C_CopyObject call as a copy of domain parameters which had its CKA_LOCAL attribute set to CK_TRUE
Notes		
	The CKA_LOCAL attribute has the value CK_TRUE if and only if the value of the domain parameters were originally generated on the token by a C_GenerateKey call.	

Table A.6: Domain Parameters Objects in PKCS # 11

Private Key Objects		
Class definition		
CK_OBJECT_CLASS	CKO_PRIVATE_KEY	
Attributes		
	Value	
Attribute	**Data Type**	**Meaning**
CKA_TOKEN	CK_BBOOL	CK_TRUE if object is a token object; CK_FALSE if object is a session object. Default is CK_FALSE.
CKA_PRIVATE	CK_BBOOL	CK_TRUE if object is a private object; CK_FALSE if object is a public object. Default value is token- specific, and may depend on the values of other attributes of the object.
CKA_MODIFIABLE	CK_BBOOL	CK_TRUE if object can be modified Default is CK_TRUE.
CKA_LABEL	RFC2279 string	Description of the object (default empty).
CKA_KEY_TYPE	CK_KEY_TYPE	Type of key
CKA_ID	Byte array	Key identifier for key (default empty)
CKA_START_DATE	CK_DATE	Start date for the key (default empty)
CKA_END_DATE	CK_DATE	End date for the key (default empty)
CKA_DERIVE	CK_BBOOL	CK_TRUE if key supports key derivation (i.e., if other keys can be derived from this one (default CK_FALSE)
CKA_LOCAL	CK_BBOOL	CK_TRUE only if key was either: - generated locally (i.e., on the token) with a C_GenerateKey or C_GenerateKeyPair call - created with a C_CopyObject call as a copy of a key which had its CKA_LOCAL attribute set to CK_TRUE
CKA_KEY_GEN_MECHANISM	CK_MECHANISM_TYPE	Identifier of the mechanism used to generate the key material.
CKA_ALLOWED_MECHANISMS	CK_MECHANISM_TYPE_PTR	A list of mechanisms allowed to be used with this key. The number of mechanisms in the array is the ulValueLen component of the attribute divided by the size of CK_MECHANISM_TYPE.
CKA_SUBJECT	Byte array	DER-encoding of certificate subject name (default empty)
CKA_SENSITIVE	CK_BBOOL	CK_TRUE if key is sensitive
CKA_DECRYPT	CK_BBOOL	CK_TRUE if key supports decryption
CKA_SIGN	CK_BBOOL	CK_TRUE if key supports signatures where the signature is an appendix to the data
CKA_SIGN_RECOVER	CK_BBOOL	CK_TRUE if key supports signatures where the data can be recovered from the signature
CKA_UNWRAP	CK_BBOOL	CK_TRUE if key supports unwrapping (i.e., can be used to unwrap other keys)
CKA_EXTRACTABLE	CK_BBOOL	CK_TRUE if key is extractable and can be wrapped
CKA_ALWAYS_SENSITIVE	CK_BBOOL	CK_TRUE if key has always had the CKA_SENSITIVE attribute set to CK_TRUE
CKA_NEVER_EXTRACTABLE	CK_BBOOL	CK_TRUE if key has never had the CKA_EXTRACTABLE attribute set to CK_TRUE
CKA_WRAP_WITH_TRUSTED	CK_BBOOL	CK_TRUE if the key can only be wrapped with a wrapping key that has CKA_TRUSTED set to CK_TRUE. Default is CK_FALSE.
CKA_UNWRAP_TEMPLATE	CK_ATTRIBUTE_ PTR	For wrapping keys. The attribute template to apply to any keys unwrapped using this wrapping key. Any user supplied template is applied after this template as if the object has already been created. The number of attributes in the array is the ulValueLen component of the attribute divided by the size of CK_ATTRIBUTE.
CKA_ALWAYS_AUTHENTICATE	CK_BBOOL	If CK_TRUE, the user has to supply the PIN for each use (sign or decrypt) with the key. Default is CK_FALSE.

Table A.7: Private Key Objects in PKCS # 11 (1 of 2)

Notes
It is intended in the interests of interoperability that the subject name and key identifier for a private key will be the same as those for the corresponding certificate and public key. However, this is not enforced by Cryptoki, and it is not required that the certificate and public key also be stored on the token.
If the CKA_SENSITIVE attribute is CK_TRUE; or if the CKA_EXTRACTABLE attribute is CK_FALSE; then certain attributes of the private key cannot be revealed in plaintext outside the token. Which attributes these are is specified for each type of private key in the attribute table in the section describing that type of key.
The CKA_ALWAYS_AUTHENTICATE attribute can be used to force re- authentication (i.e. force the user to provide a PIN) for each use of a private key. "Use" in this case means a cryptographic operation such as sign or decrypt. This attribute may only be set to CK_TRUE when CKA_PRIVATE is also CK_TRUE.
Re-authentication occurs by calling C_Login with userType set to CKU_CONTEXT_SPECIFIC immediately after a cryptographic operation using the key has been initiated (e.g. after C_SignInit). In this call, the actual user type is implicitly given by the usage requirements of the active key. If C_Login returns CKR_OK the user was successfully authenticated and this sets the active key in an authenticated state that lasts until the cryptographic operation has successfully or unsuccessfully been completed (e.g. by C_Sign, C_SignFinal...). A return value CKR_PIN_INCORRECT from C_Login means that the user was denied permission to use the key and continuing the cryptographic operation will result in a behavior as if C_Login had not been called. In both of these cases the session state will remain the same, however repeated failed re-authentication attempts may cause the PIN to be locked. C_Login returns in this case CKR_PIN_LOCKED and this also logs the user out from the token. Failing or omitting to re-authenticate when CKA_ALWAYS_AUTHENTICATE is set to CK_TRUE will result in CKR_USER_NOT_LOGGED_IN to be returned from calls using the key. C_Login will return CKR_OPERATION_NOT_INITIALIZED, but the active cryptographic operation will not be affected, if an attempt is made to re-authenticate when CKA_ALWAYS_AUTHENTICATE is set to CK_FALSE.

Table A.8: Private Key Objects in PKCS # 11 (2 of 2)

Public Key Objects		
Class definition		
CK_OBJECT_CLASS	CKO_PUBLIC_KEY	
Attributes		
Value		
Attribute	**Data Type**	**Meaning**
CKA_TOKEN	CK_BBOOL	CK_TRUE if object is a token object; CK_FALSE if object is a session object. Default is CK_FALSE.
CKA_PRIVATE	CK_BBOOL	CK_TRUE if object is a private object; CK_FALSE if object is a public object. Default value is token- specific, and may depend on the values of other attributes of the object.
CKA_MODIFIABLE	CK_BBOOL	CK_TRUE if object can be modified Default is CK_TRUE.
CKA_LABEL	RFC2279 string	Description of the object (default empty).
CKA_KEY_TYPE	CK_KEY_TYPE	Type of key
CKA_ID	Byte array	Key identifier for key (default empty)
CKA_START_DATE	CK_DATE	Start date for the key (default empty)
CKA_END_DATE	CK_DATE	End date for the key (default empty)
CKA_DERIVE	CK_BBOOL	CK_TRUE if key supports key derivation (i.e., if other keys can be derived from this one (default CK_FALSE)
CKA_LOCAL	CK_BBOOL	CK_TRUE only if key was either: - generated locally (i.e., on the token) with a C_GenerateKey or C_GenerateKeyPair call - created with a C_CopyObject call as a copy of a key which had its CKA_LOCAL attribute set to CK_TRUE
CKA_KEY_GEN_MECHANISM	CK_MECHANISM_TYPE	Identifier of the mechanism used to generate the key material.
CKA_ALLOWED_MECHANISMS	CK_MECHANISM_TYPE_PTR	A list of mechanisms allowed to be used with this key. The number of mechanisms in the array is the ulValueLen component of the attribute divided by the size of CK_MECHANISM_TYPE.
CKA_SUBJECT	Byte array	DER-encoding of the key subject name (default empty)
CKA_ENCRYPT	CK_BBOOL	CK_TRUE if key supports encryption
CKA_VERIFY	CK_BBOOL	CK_TRUE if key supports verification where the signature is an appendix to the data
CKA_VERIFY_RECOVER	CK_BBOOL	CK_TRUE if key supports verification where the data is recovered from the signature
CKA_WRAP	CK_BBOOL	CK_TRUE if key supports wrapping (i.e., can be used to wrap other keys)
CKA_TRUSTED	CK_BBOOL	The key can be trusted for the application that it was created. The wrapping key can be used to wrap keys with CKA_WRAP_WITH_TRUSTED set to CK_TRUE.
CKA_WRAP_TEMPLATE	CK_ATTRIBUTE_ PTR	For wrapping keys. The attribute template to match against any keys wrapped using this wrapping key. Keys that do not match cannot be wrapped. The number of attributes in the array is the ulValueLen component of the attribute divided by the size of CK_ATTRIBUTE.

Table A.9: Public Key Objects in PKCS # 11 (1 of 2)

Notes	
	It is intended in the interests of interoperability that the subject name and key identifier for a public key will be the same as those for the corresponding certificate and private key. However, Cryptoki does not enforce this, and it is not required that the certificate and private key also be stored on the token.

Mapping of X.509 key usage flags to cryptoki:	
keys in X.509 public key	**Corresponding cryptoki attributes for public keys**
dataEncipherment	CKA_ENCRYPT
keyCertSign, cRLSign	CKA_VERIFY
keyCertSign, cRLSign	CKA_VERIFY_RECOVER
keyAgreement	CKA_DERIVE
keyEncipherment	CKA_WRAP
nonRepudiation	CKA_VERIFY
nonRepudiation	CKA_VERIFY_RECOVER

Table A.10: Public Key Objects in PKCS # 11 (2 of 2)

Secret Key Objects		
Class definition		
CK_OBJECT_CLASS	CKO_SECRET_KEY	
Attributes		
	Value	
Attribute	**Data Type**	**Meaning**
CKA_TOKEN	CK_BBOOL	CK_TRUE if object is a token object; CK_FALSE if object is a session object. Default is CK_FALSE.
CKA_PRIVATE	CK_BBOOL	CK_TRUE if object is a private object; CK_FALSE if object is a public object. Default value is token- specific, and may depend on the values of other attributes of the object.
CKA_MODIFIABLE	CK_BBOOL	CK_TRUE if object can be modified Default is CK_TRUE.
CKA_LABEL	RFC2279 string	Description of the object (default empty).
CKA_KEY_TYPE	CK_KEY_TYPE	Type of key
CKA_ID	Byte array	Key identifier for key (default empty)
CKA_START_DATE	CK_DATE	Start date for the key (default empty)
CKA_END_DATE	CK_DATE	End date for the key (default empty)
CKA_DERIVE	CK_BBOOL	CK_TRUE if key supports key derivation (i.e., if other keys can be derived from this one (default CK_FALSE)
CKA_LOCAL	CK_BBOOL	CK_TRUE only if key was either: - generated locally (i.e., on the token) with a C_GenerateKey or C_GenerateKeyPair call - created with a C_CopyObject call as a copy of a key which had its CKA_LOCAL attribute set to CK_TRUE
CKA_KEY_GEN_MECHANISM	CK_MECHANISM_TYPE	Identifier of the mechanism used to generate the key material.
CKA_ALLOWED_MECHANISMS	CK_MECHANISM_TYPE_PTR	A list of mechanisms allowed to be used with this key. The number of mechanisms in the array is the ulValueLen component of the attribute divided by the size of CK_MECHANISM_TYPE.
CKA_SENSITIVE	CK_BBOOL	CK_TRUE if object is sensitive (default CK_FALSE)
CKA_ENCRYPT	CK_BBOOL	CK_TRUE if key supports encryption
CKA_DECRYPT	CK_BBOOL	CK_TRUE if key supports decryption
CKA_SIGN	CK_BBOOL	CK_TRUE if key supports signatures (i.e., authentication codes) where the signature is an appendix to the data
CKA_VERIFY	CK_BBOOL	CK_TRUE if key supports verification (i.e., of authentication codes) where the signature is an appendix to the data
CKA_WRAP	CK_BBOOL	CK_TRUE if key supports wrapping (i.e., can be used to wrap other keys)
CKA_UNWRAP	CK_BBOOL	CK_TRUE if key supports unwrapping (i.e., can be used to unwrap other keys)
CKA_EXTRACTABLE	CK_BBOOL	CK_TRUE if key is extractable and can be wrapped
CKA_ALWAYS_SENSITIVE	CK_BBOOL	CK_TRUE if key has always had the CKA_SENSITIVE attribute set to CK_TRUE
CKA_NEVER_EXTRACTABLE	CK_BBOOL	CK_TRUE if key has never had the CKA_EXTRACTABLE attribute set to CK_TRUE
CKA_CHECK_VALUE	Byte array	Key checksum
CKA_WRAP_WITH_TRUSTED	CK_BBOOL	CK_TRUE if the key can only be wrapped with a wrapping key that has CKA_TRUSTED set to CK_TRUE. Default is CK_FALSE.
CKA_TRUSTED	CK_BBOOL	The wrapping key can be used to wrap keys with CKA_WRAP_WITH_TRUSTED set to CK_TRUE.
CKA_WRAP_TEMPLATE	CK_ATTRIBUTE_ PTR	For wrapping keys. The attribute template to match against any keys wrapped using this wrapping key. Keys that do not match cannot be wrapped. The number of attributes in the array is the ulValueLen component of the attribute divided by the size of CK_ATTRIBUTE
CKA_UNWRAP_TEMPLATE	CK_ATTRIBUTE_ PTR	For wrapping keys. The attribute template to apply to any keys unwrapped using this wrapping key. Any user supplied template is applied after this template as if the object has already been created. The number of attributes in the array is the ulValueLen component of the attribute divided by the size of CK_ATTRIBUTE.

Table A.11: Secret Key Objects in PKCS # 11 (1 of 2)

Notes	
	If the CKA_SENSITIVE attribute is CK_TRUE, or if the CKA_EXTRACTABLE attribute is CK_FALSE, then certain attributes of the secret key cannot be revealed in plaintext outside the token. Which attributes these are is specified for each type of secret key in the attribute table in the section describing that type of key.
	The key check value (KCV) attribute for symmetric key objects to be called CKA_CHECK_VALUE, of type byte array, length 3 bytes, operates like a fingerprint, or checksum of the key. They are intended to be used to cross-check symmetric keys against other systems where the same key is shared, and as a validity check after manual key entry or restore from backup. Refer to object definitions of specific key types for KCV algorithms.
	Properties: - For two keys that are cryptographically identical the value of this attribute should be identical. - CKA_CHECK_VALUE should not be usable to obtain any part of the key value. - Non-uniqueness. Two different keys can have the same CKA_CHECK_VALUE. This is unlikely (the probability can easily be calculated) but possible.
	The attribute is optional but if supported the value of the attribute is always supplied by the library regardless of how the key object is created or derived. It shall be supplied even if the encryption operation for the key is forbidden (i.e. when CKA_ENCRYPT is set to CK_FALSE).
	If a value is supplied in the application template (allowed but never necessary) then, if supported, it must match what the library calculates it to be or the library returns a CKR_ATTRIBUTE_VALUE_INVALID. If the library does not support the attribute then it should ignore it. Allowing the attribute in the template this way does no harm and allows the attribute to be treated like any other attribute for the purposes of key wrap and unwrap where the attributes are preserved also.
	The generation of the KCV may be prevented by the application supplying the attribute in the template as a no-value (0 length) entry. The application can query the value at any time like any other attribute using C_GetAttributeValue. C_SetAttributeValue may be used to destroy the attribute, by supplying no-value.
	Unless otherwise specified for the object definition, the value of this attribute is derived from the key object by taking the first three bytes of an encryption of a single block of null (0x00) bytes, using the default cipher and mode (e.g. ECB) associated with the key type of the secret key object.

Table A.12: Secret Key Objects in PKCS # 11 (2 of 2)

Certificate Objects				
Class definition				
	CKA_OBJECT_CLASS		CKO_CERTIFICATE	
Common Attributes	**Attributes**	**Data Type**	**Meaning**	
	CKA_TOKEN	CK_BBOOL	CK_TRUE if object is a token object; CK_FALSE if object is a session object. Default is CK_FALSE.	
	CKA_PRIVATE	CK_BBOOL	CK_TRUE if object is a private object; CK_FALSE if object is a public object. Default value is token- specific, and may depend on the values of other attributes of the object.	
	CKA_MODIFIABLE	CK_BBOOL	CK_TRUE if object can be modified Default is CK_TRUE.	
	CKA_LABEL	RFC2279 string	Description of the object (default empty).	
	CKA_CERTIFICATE_TYPE	CK_CERTIFICATE_TYPE	Type of certificate	
	CKA_TRUSTED	CK_BBOOL	The certificate can be trusted for the application that it was created.	
	CKA_CERTIFICATE_CATEGORY	CK_ULONG	Categorization of the certificate: 0 = unspecified (default value), 1 = token user, 2 = authority, 3 = other entity	
	CKA_CHECK_VALUE	Byte array	Checksum	
	CKA_START_DATE	CK_DATE	Start date for the certificate (default empty)	
	CKA_END_DATE	CK_DATE	End date for the certificate (default empty)	

Attributes			
CK_CERTIFICATE_TYPE	**Value**		
	Attribute	**Data Type**	**Meaning**
CKC_X_509	CKA_SUBJECT	Byte array	DER-encoding of the certificate subject name
	CKA_ID	Byte array	Key identifier for public/private key pair (default empty)
	CKA_ISSUER	Byte array	DER-encoding of the certificate issuer name (default empty)
	CKA_SERIAL_NUMBER	Byte array	DER-encoding of the certificate serial number (default empty)
	CKA_VALUE	Byte array	BER-encoding of the certificate
	CKA_URL	RFC2279 string	If not empty this attribute gives the URL where the complete certificate can be obtained (default empty)
	CKA_HASH_OF_SUBJECT_PUBLIC_KEY	Byte array	SHA-1 hash of the subject public key (default empty)
	CKA_HASH_OF_ISSUER_PUBLIC_KEY	Byte array	SHA-1 hash of the issuer public key (default empty)
	CKA_JAVA_MIDP_SECURITY_DOMAIN	CK_ULONG	Java MIDP security domain: 0 = unspecified (default value), 1 = manufacturer, 2 = operator, 3 = third party
CKC_WTLS	CKA_SUBJECT	Byte array	WTLS-encoding (Identifier type) of the certificate subject
	CKA_ISSUER	Byte array	WTLS-encoding (Identifier type) of the certificate issuer (default empty)
	CKA_VALUE	Byte array	WTLS-encoding of the certificate
	CKA_URL	RFC2279 string	If not empty this attribute gives the URL where the complete certificate can be obtained
	CKA_HASH_OF_SUBJECT_PUBLIC_KEY	Byte array	SHA-1 hash of the subject public key (default empty)
	CKA_HASH_OF_ISSUER_PUBLIC_KEY	Byte array	SHA-1 hash of the issuer public key (default empty)
CKC_X_509_ATTR_CERT	CKA_OWNER	Byte Array	DER-encoding of the attribute certificate's subject field. This is distinct from the CKA_SUBJECT attribute contained in CKC_X_509 certificates because the ASN.1 syntax and encoding are different.
	CKA_AC_ISSUER	Byte Array	DER-encoding of the attribute certificate's issuer field. This is distinct from the CKA_ISSUER attribute contained in CKC_X_509 certificates because the ASN.1 syntax and encoding are different. (default empty)
	CKA_SERIAL_NUMBER	Byte Array	DER-encoding of the certificate serial number. (default empty)

Table A.13: Certificate Objects in PKCS # 11 (1 of 3)

CKC_X_509_ATTR_CERT	CKA_ATTR_TYPES	Byte Array	BER-encoding of a sequence of object identifier values corresponding to the attribute types contained in the certificate. When present, this field offers an opportunity for applications to search for a particular attribute certificate without fetching and parsing the certificate itself. (default empty)
	CKA_VALUE	Byte Array	BER-encoding of the certificate.
	Notes		
Certificate objects	The CKA_CERTIFICATE_TYPE attribute may not be modified after an object is created.		
	This version of Cryptoki supports the following certificate types: - X.509 public key certificate - WTLS public key certificate		
	The CKA_TRUSTED attribute cannot be set to CK_TRUE by an application. It must be set by a token initialization application or by the token's SO. Trusted certificates cannot be modified.		
	The CKA_CERTIFICATE_CATEGORY attribute is used to indicate if a stored certificate is a user certificate for which the corresponding private key is available on the token ("token user"), a CA certificate ("authority"), or an other end-entity certificate ("other entity"). This attribute may not be modified after an object is created.		
	The CKA_CERTIFICATE_CATEGORY and CKA_TRUSTED attributes will together be used to map to the categorization of the certificates. A certificate in the certificates CDF will be marked with category "token user". A certificate in the trustedCertificates CDF or in the usefulCertificates CDF will be marked with category "authority" or "other entity" depending on the CommonCertificateAttribute.authority attribute and the CKA_CHECK_VALUE: The value of this attribute is derived from the certificate by taking the first three bytes of the SHA-1 hash of the certificate object's CKA_VALUE attribute.		
	The CKA_START_DATE and CKA_END_DATE attributes are for reference only; Cryptoki does not attach any special meaning to them. When present, the application is responsible to set them to values that match the certificate's encoded "not before" and "not after" fields (if any).		
X.509 Certificate Object Attributes	Only the CKA_ID, CKA_ISSUER, and CKA_SERIAL_NUMBER attributes may be modified after the object is created.		
	The CKA_ID attribute is intended as a means of distinguishing multiple public- key/private-key pairs held by the same subject (whether stored in the same token or not). (Since the keys are distinguished by subject name as well as identifier, it is possible that keys for different subjects may have the same CKA_ID value without introducing any ambiguity.) It is intended in the interests of interoperability that the subject name and key identifier for a certificate will be the same as those for the corresponding public and private keys (though it is not required that all be stored in the same token). However, Cryptoki does not enforce this association, or even the uniqueness of the key identifier for a given subject; in particular, an application may leave the key identifier empty.		
	The CKA_ISSUER and CKA_SERIAL_NUMBER attributes are for compatibility with PKCS #7 and Privacy Enhanced Mail (RFC1421). Note that with the version 3 extensions to X.509 certificates, the key identifier may be carried in the certificate. It is intended that the CKA_ID value be identical to the key identifier in such a certificate extension, although this will not be enforced by Cryptoki.		
	The CKA_URL attribute enables the support for storage of the URL where the certificate can be found instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile environments.		
	The CKA_HASH_OF_SUBJECT_PUBLIC_KEY and CKA_HASH_OF_ISSUER_PUBLIC_KEY attributes are used to store the hashes of the public keys of the subject and the issuer. They are particularly important when only the URL is available to be able to correlate a certificate with a private key and when searching for the certificate of the issuer.		
	The CKA_JAVA_MIDP_SECURITY_DOMAIN attribute associates a certificate with a Java MIDP security domain.		
WTLS public key certificate objects	Only the CKA_ISSUER attribute may be modified after the object has been created.		
	The encoding for the CKA_SUBJECT, CKA_ISSUER, and CKA_VALUE attributes can be found in [WTLS]		
	The CKA_URL attribute enables the support for storage of the URL where the certificate can be found instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile environments.		
	The CKA_HASH_OF_SUBJECT_PUBLIC_KEY and CKA_HASH_OF_ISSUER_PUBLIC_KEY attributes are used to store the hashes of the public keys of the subject and the issuer. They are particularly important when only the URL is available to be able to correlate a certificate with a private key and when searching for the certificate of the issuer.		
X.509 attribute certificate objects	Only the CKA_AC_ISSUER, CKA_SERIAL_NUMBER and CKA_ATTR_TYPES attributes may be modified after the object is created.		

Table A.14: Certificate Objects in PKCS # 11 (2 of 3)

	Sample Templates

X.509 Certificate Object Attributes	CK_OBJECT_CLASS class = CKO_CERTIFICATE;
	CK_CERTIFICATE_TYPE certType = CKC_X_509;
	CK_UTF8CHAR label[] = "A certificate object";
	CK_BYTE subject[] = {...};
	CK_BYTE id[] = {123};
	CK_BYTE certificate[] = {...};
	CK_BBOOL true = CK_TRUE;
	CK_ATTRIBUTE template[] = {
	{CKA_CLASS, &class, sizeof(class)},
	{CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)};
	{CKA_TOKEN, &true, sizeof(true)},
	{CKA_LABEL, label, sizeof(label)-1},
	{CKA_SUBJECT, subject, sizeof(subject)},
	{CKA_ID, id, sizeof(id)},
	{CKA_VALUE, certificate, sizeof(certificate)}
	};

WTLS public key certificate objects	CK_OBJECT_CLASS class = CKO_CERTIFICATE;
	CK_CERTIFICATE_TYPE certType = CKC_WTLS;
	CK_UTF8CHAR label[] = "A certificate object";
	CK_BYTE subject[] = {...};
	CK_BYTE certificate[] = {...};
	CK_BBOOL true = CK_TRUE;
	CK_ATTRIBUTE template[] = {
	{CKA_CLASS, &class, sizeof(class)},
	{CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)};
	{CKA_TOKEN, &true, sizeof(true)},
	{CKA_LABEL, label, sizeof(label)-1},
	{CKA_SUBJECT, subject, sizeof(subject)},
	{CKA_VALUE, certificate, sizeof(certificate)}
	};

X.509 attribute certificate objects	CK_OBJECT_CLASS class = CKO_CERTIFICATE;
	CK_CERTIFICATE_TYPE certType = CKC_X_509_ATTR_CERT;
	CK_UTF8CHAR label[] = "An attribute certificate object";
	CK_BYTE owner[] = {...};
	CK_BYTE certificate[] = {...};
	CK_BBOOL true = CK_TRUE;
	CK_ATTRIBUTE template[] = {
	{CKA_CLASS, &class, sizeof(class)},
	{CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)};
	{CKA_TOKEN, &true, sizeof(true)},
	{CKA_LABEL, label, sizeof(label)-1},
	{CKA_OWNER, owner, sizeof(owner)},
	{CKA_VALUE, certificate, sizeof(certificate)}
	};

Table A.15: Certificate Objects in PKCS # 11 (3 of 3)

A.3 Plugin Installed in Several Platforms

The following images show our plugin (*SmartCardsEveryWhere*) being listed as one of the available plugins in each one of the web browsers: Google Chrome, Mozilla Firefox, and Microsoft Internet Explorer. We present an image for each supported OS as well.

Google Chrome

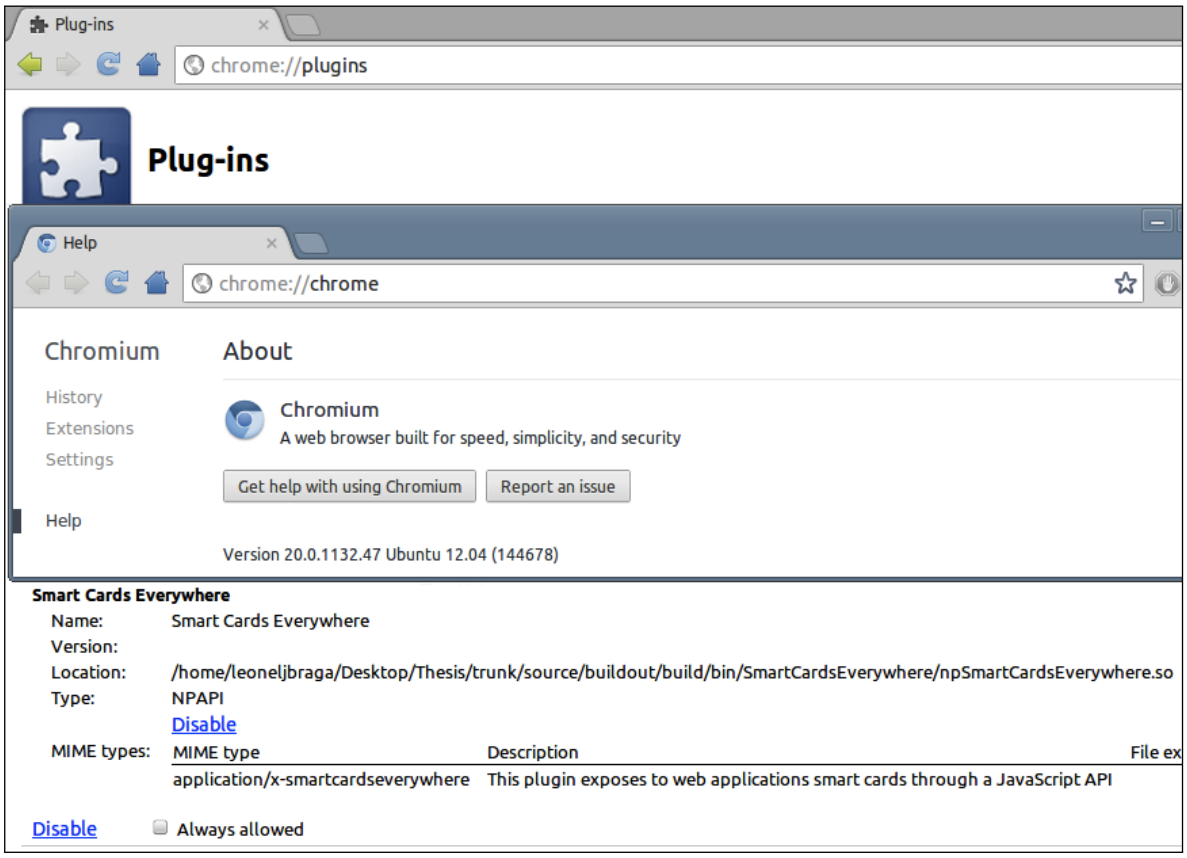


Figure A.1: Plugin Installed in the LUbuntu version of Google Chrome

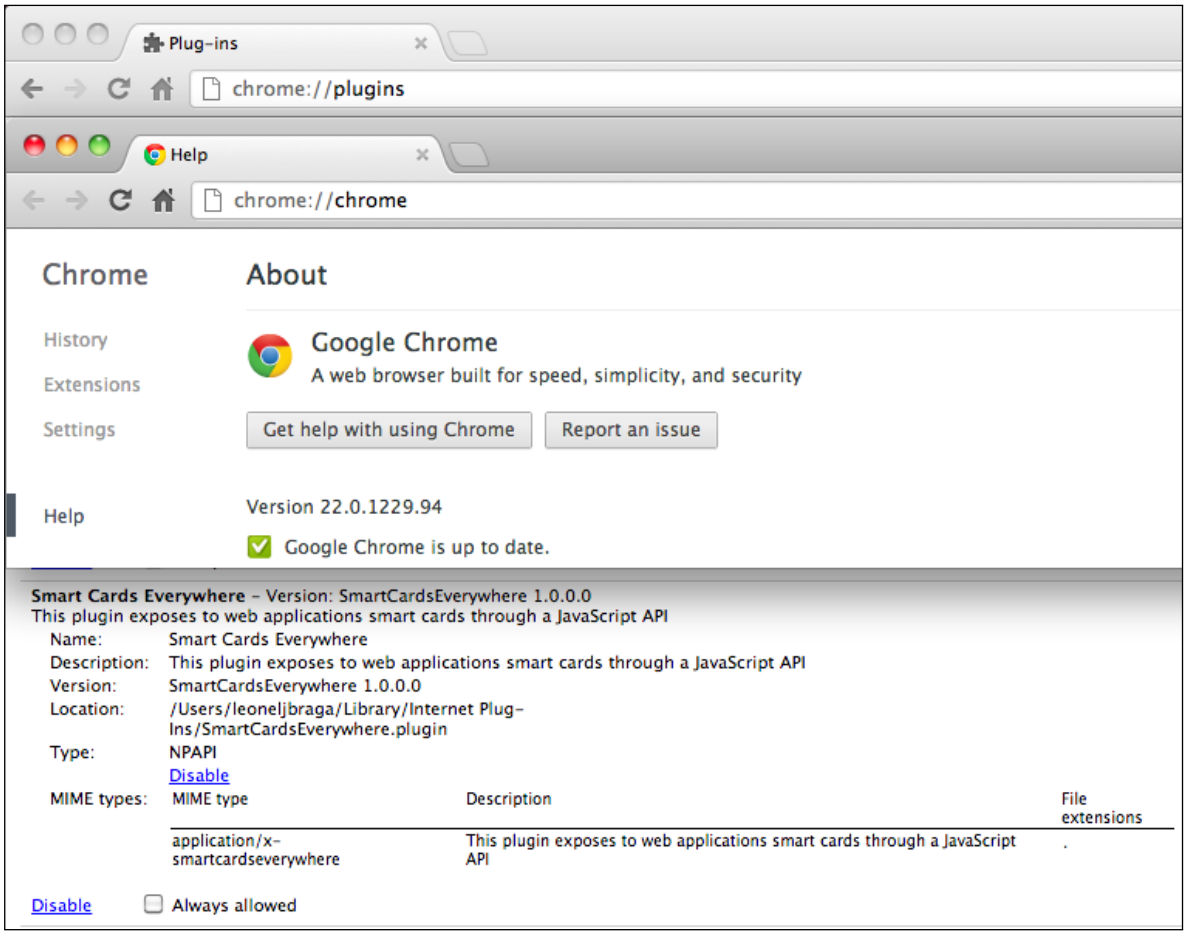


Figure A.2: Plugin Installed in the Mac OS X version of Google Chrome

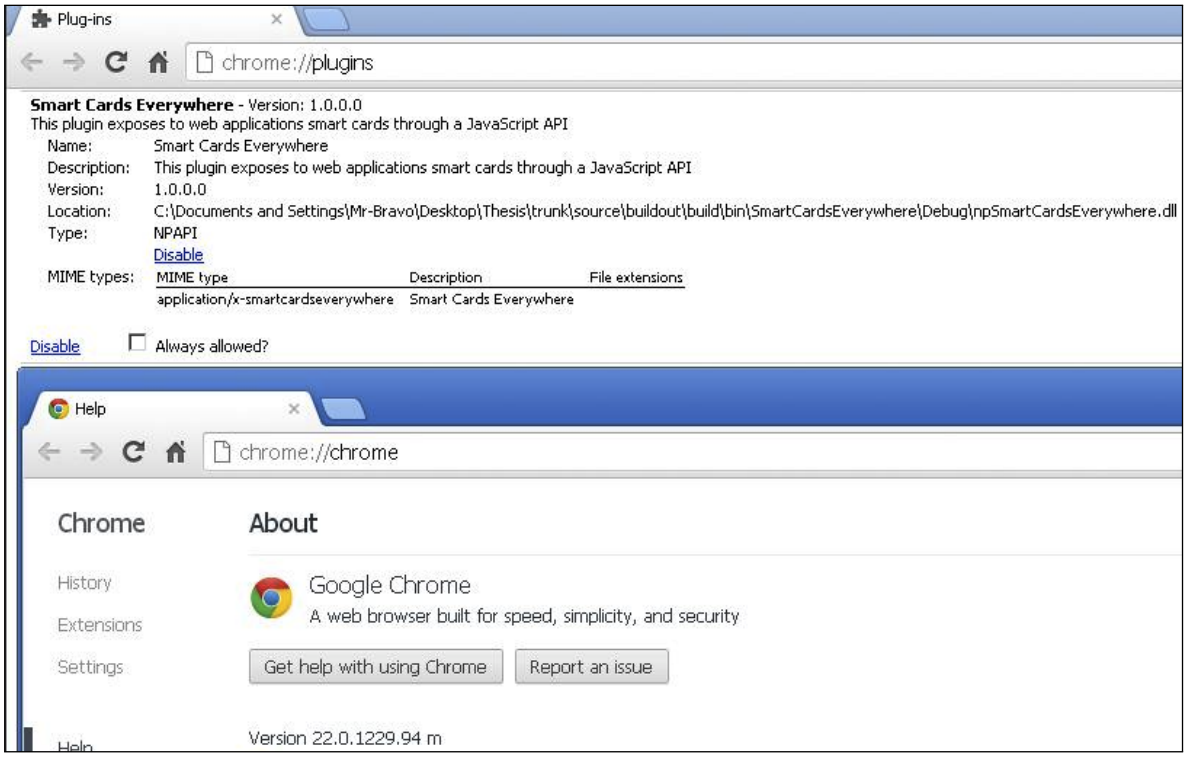


Figure A.3: Plugin Installed in the Microsoft Windows version of Google Chrome

Mozilla Firefox

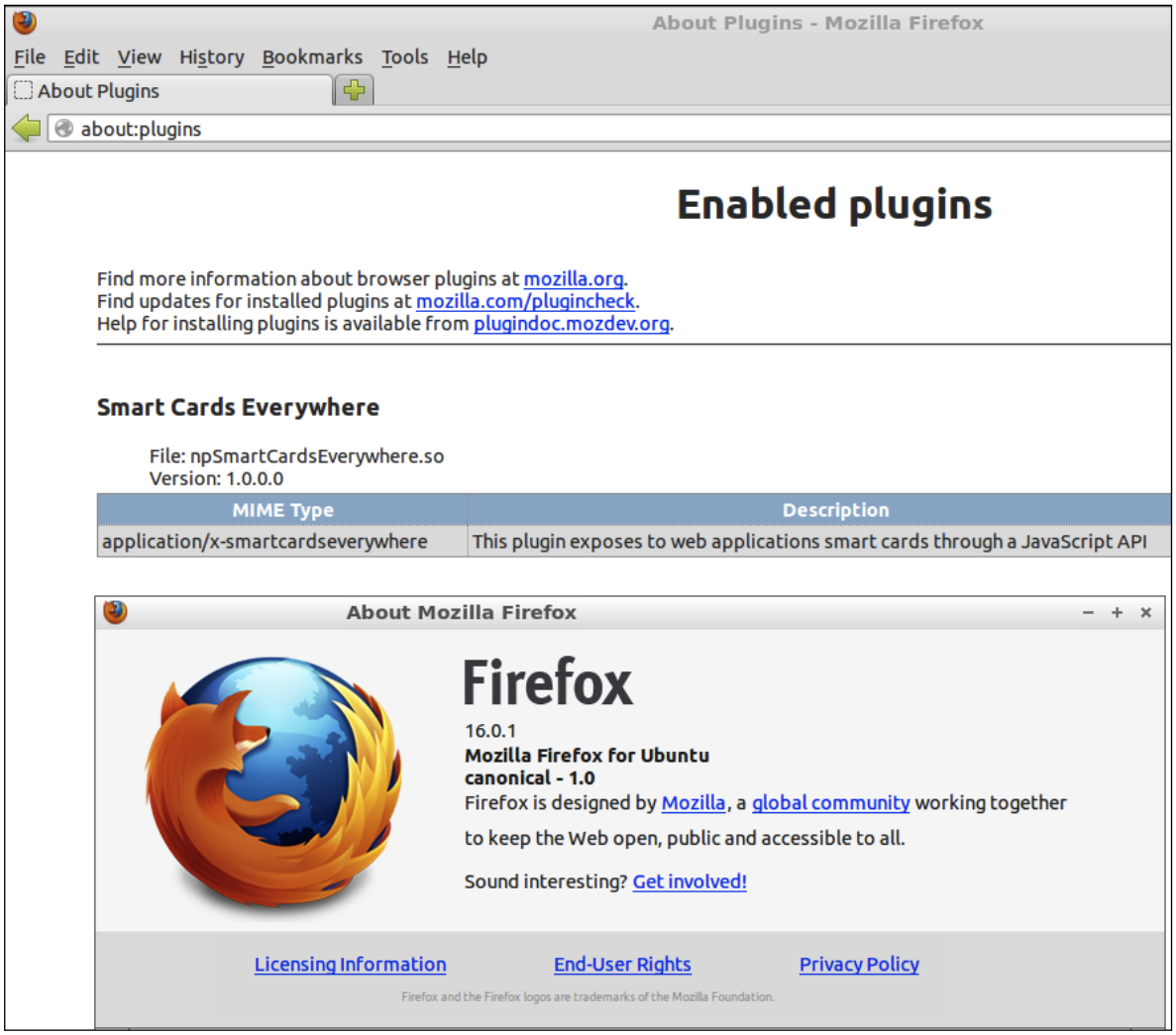


Figure A.4: Plugin Installed in LUbuntu version of Mozilla Firefox

Figure A.5: Plugin Installed in the Mac OS X version of Mozilla Firefox

Figure A.6: Plugin Installed in the Microsoft Windows version of Mozilla Firefox

Microsoft Internet Explorer

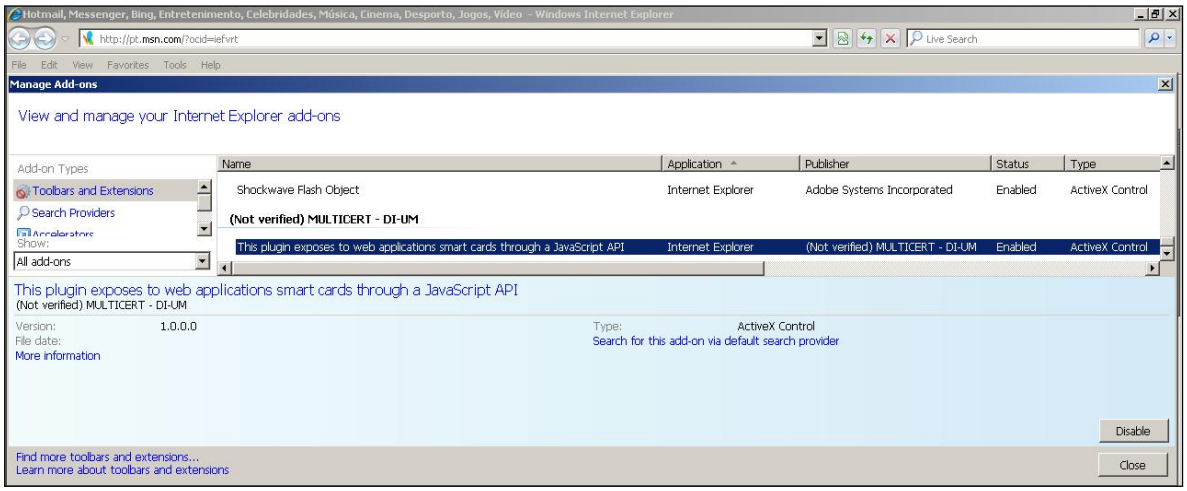


Figure A.7: Plugin Installed in Microsoft Internet Explorer